Chapter 3

Processor Cores

Before considering systems based on multi-core chips, we first consider the microarchitecture of individual processor cores. We will consider both simple cores that issue and execute instructions in the architected program sequence (in-order), as well as more advanced superscalar cores capable of issuing multiple instructions per cycle, out-of-order with to the architected sequence. Demand for performance has driven single core designs steadily in the direction of powerful uniprocessors, toward wide superscalar processors, in particular. Superscalar cores are dominant in both client and server computers. While this trend continues, it has slowed substantially in recent years. Power considerations and the ability to enhance throughput with multiple cores has motivated a re-thinking of simple in-order cores in high performance systems. Consequently both in- and out-of-order cores are likely to be used in future systems.

An important technique used in some multi-core systems is *multi-threading* where a single hardware core can support multiple threads of execution. This is done by implementing hardware that holds multiple sets of architected state (program counters and registers), and sharing a core's hardware resources among the multiple threads. In essence, multi-threading allows a single hardware core to have many features of a multiprocessor; to software there is no logical difference. Because of our interest in multiprocessor systems, we will provide an overview of conventional processor cores, and then pay special attention to multi-threaded cores.

3.1 In-Order Cores

Figure 1 illustrates a simple in-order pipeline where instructions pass through a series of stages as they are executed. A pipeline performs its operations in an assembly-line like fashion. This allows multiple instructions to be in the pipeline simultaneously, though at most one instruction is in each pipeline stage at any given time. In Figure 1 instructions pass through the pipeline in their natural, architected program sequence (i.e., in-order). First, following the sequence of addresses provided by the program counter, instructions are fetched and decoded. Then, assuming the instructions' input operands are ready, the operands are read either from registers or from memory, and the instructions are executed in execution units such as adders, shifters, and multipliers. Finally, instruction results are written to registers or stored in memory.

Some instructions depend on results of preceding instructions, and when the results are not ready as inputs when an instruction needs them, the pipeline must *stall* the instruction in its pipeline stage until the preceding instruction produces its needed result. If an instruction takes several cycles to produce its results, e.g., if it misses in the L1 data cache and loads data from an L2 cache, then a following dependent instruction may suffer a number of stall cycles. In general, it should be apparent that in an in-order pipeline the sequence in which instructions appear and their data dependences influence the number of stalls, and therefore, overall pipeline performance.

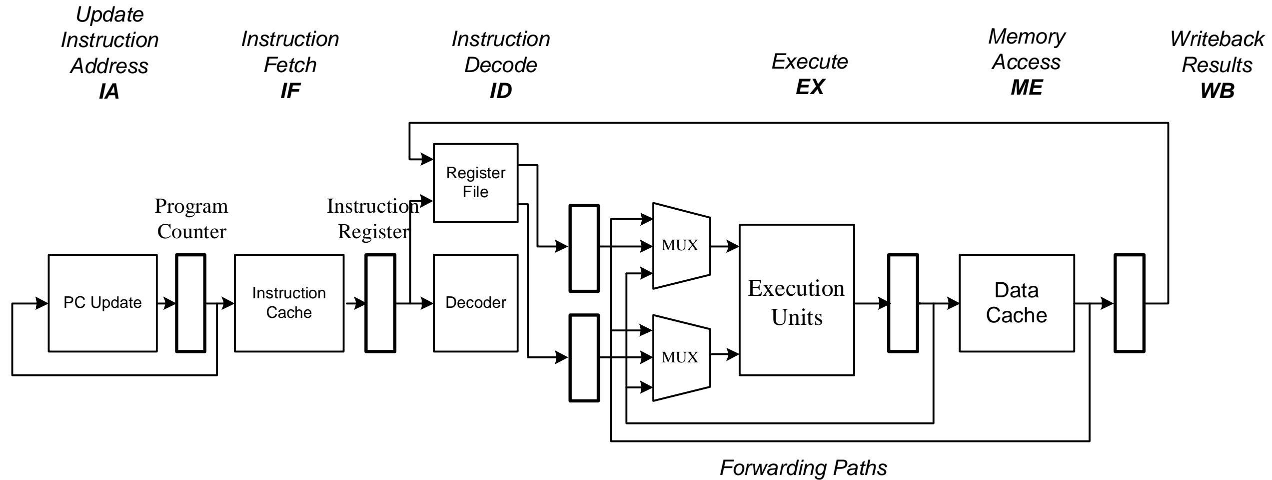
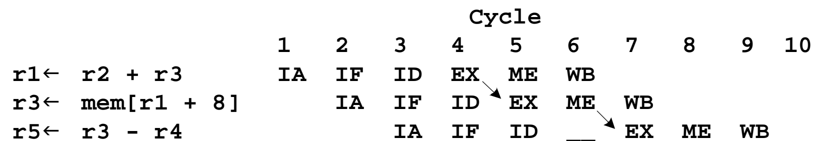


Figure 1. Pipelined, in-order processor microarchitecture. This pipeline, when drawn without the IA stage, is commonly referred to as a "Five-Stage Pipeline".

To reduce performance penalties due to stalls, pipelines employ forwarding paths. An example that illustrates stalls and forwarding is in Figure 2. On the left side of Figure 2, three machine level instructions are shown using a register transfer notation. The first instruction adds registers r1 and r3, placing the result into register r1. The second instruction, a load from memory, uses the value in register r1 as an input operand, so it is data dependent on the first instruction. The third instruction, a subtract, is dependent on the second for the operand register r3. To the right of the instructions, pipeline processing is illustrated with time in clock cycles running from left to right. During each cycle shown, the pipeline stage that holds a given instruction is shown. The first instruction proceeds down the pipeline, visiting a new stage each cycle. The second instruction consumes the result of the first. This value is forwarded from the first instruction to the second via a forwarding path from the end of the EX stage back to the beginning of the EX stage; this forwarding path is shown in Figure 1. The third instruction uses the result of the second, but the value isn't ready at the time the third instruction needs it, so it stalls for one cycle. Then, the value is forwarded from the end of the ME stage to the EX stage. If the second instruction had missed in the data cache, then the third instruction would have stalled for many more cycles, while the missed data was retrieved from lower levels in the memory hierarchy.



	Cycle									
	1	2	3	4	5	6	7	8	9	10
r1← r2 + r3	IA	IF	ID	EX	ME	WB				
r3← mem[r1 + 8]		IA	IF	ID	EX	ME	WB			
r5← r3 - r4			IA	IF	ID	__	EX	ME	WB	

Figure 2. Three instructions flow down a pipeline; the first forwards data to the second (as shown with an arrow). The second instruction first stalls for one cycle, then forwards data to the third.

The pipeline illustrated in Figure 1 is commonly used for register-register instruction sets, where an instruction may either access memory or operate on register values, but not both. This is a feature of most RISCs (reduced instruction set computers). A typical pipeline for a CISC (complex instruction set computer), using a register-storage instruction set, contains a pipeline stage for address generation that precedes the data cache, and an execute stage that follows the data cache. Many recent instruction sets, for example PowerPC and SPARC, employ a RISC style, while older instruction sets, such as the IBM 360/370 and Intel x86 employ a register-storage CISC instruction set.

© 2007 by J. E. Smith

A class of more advanced in-order processors is illustrated in Figure 3; these processors can execute multiple instructions per cycle, but only in the original compiled sequence. It is up to the compiler to arrange and combine multiple parallel (independent) instructions into a very long instruction word (VLIW). The in-order VLIW approach simplifies the instruction issue logic, compared to an out-of-order superscalar processor, but it puts the burden of finding and re-ordering groups of independent instructions on the compiler rather than the hardware as in superscalar out-of-order machines. In current general purpose computers, this in-order, wide word approach is only used in processors implementing the Intel IPF (formerly IA-64) instruction set. Hence, when we discuss core microarchitectures, we will focus more on simple single issue processors and out-of-order superscalar cores.

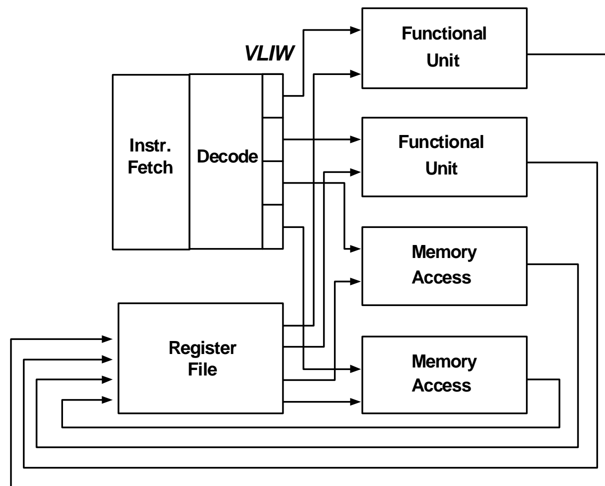


Figure 3. A VLIW microarchitecture

3.2 Out-of-Order Cores

High performance superscalar cores issue multiple instructions in a cycle, and can reorder instructions dynamically [2]. A superscalar microarchitecture is shown in Figure 4. In a superscalar processor, more than one instruction can be fetched, decoded, executed, and committed in a single clock cycle. Thus, the peak instruction throughput is increased significantly when compared with the simple pipeline as in Figure 1.

After decoding, instructions are dispatched into an instruction issue buffer. From the issue buffer, instructions can issue and begin execution when their input operands are ready, without regard to the original program sequence, i.e., they can issue "out-of-order". This avoids many of the stalls that occur in an in-order pipeline when instructions depend on preceding instructions. The superscalar processor illustrated in Figure 4 is a generic design, similar in style to many processors in use today. A modern superscalar processor is often described in terms of an "in-order" frontend unit that fetches and decodes instructions in-order and places them into an instruction issue buffer. Then, there is an "out-of-order" backend where data-ready instructions are issued from the issue buffer and executed. We will discuss the front- and back-ends briefly in following subsections. We will then separately discuss the data cache and its associated interface, largely consists of address and data buffering. It is this latter portion of the design that will be of most interest to us when discussing multiprocessor implementations.

© 2007 by J. E. Smith

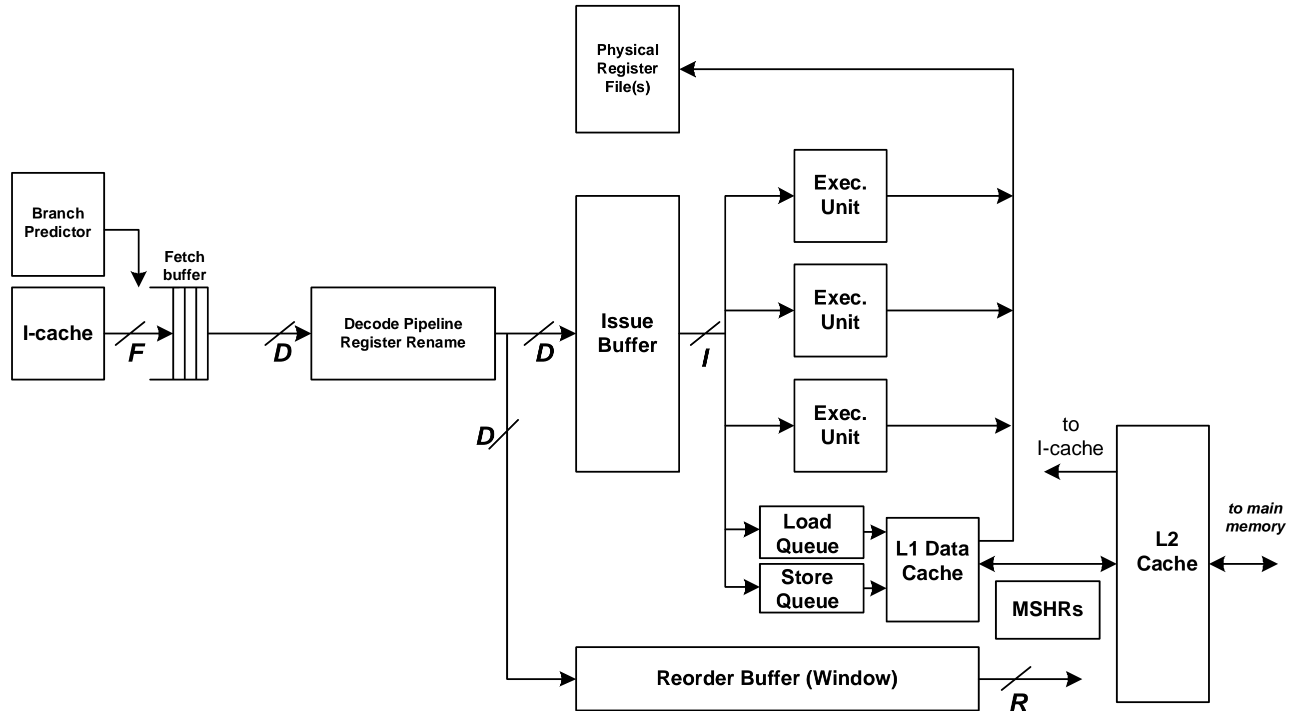


Figure 4. A superscalar microarchitecture.

3.2.1 Superscalar Front-End

At the left of Figure 4 is the instruction delivery subsystem which combines elements of instruction fetch, instruction buffering, and branch prediction. It is important that the instruction delivery subsystem provides a sustained flow of instructions that matches the capacity of the rest of the processor to issue, execute, and commit instructions. Depending on the required instruction delivery rate, the fetch unit may be designed with a certain level of aggressiveness; for example, it might fetch up to the first branch, to the first not taken branch, or past multiple branches, both taken and not-taken. Because of the variability in software basic block sizes, the peak fetch width, denoted as F in Figure 4, will typically be greater than the pipeline width (denoted as D), with an instruction fetch buffer to moderate the flow into the pipeline so that an average pipeline rate of D instructions per cycle can be sustained.

After instructions are fetched, they are decoded, their registers are renamed, and they are dispatched into an instruction issue buffer and a reorder buffer (ROB). The maximum dispatch width is D instructions per cycle. In a modern superscalar processor, the instruction issue buffer is separate from the ROB. The role of the ROB is to maintain the architected instruction sequence of all active, or "in flight" instructions so that the correct process state can be restored in the event of a trap or a branch misprediction. The issue buffer contains only the non-issued instructions -- the instructions waiting for their dependences to be clear so that they can issue. From this perspective, implementing an instruction issue buffer that is separate from the reorder buffer is an optimization providing expensive issue logic resources only on those instructions that have not yet issued. Some early superscalar processor designs, for example the HP PA-8000 [31], did not use this optimization, but instead issued instructions directly from the ROB.

© 2007 by J. E. Smith

3.2.2 Superscalar Backend

Separating the pipeline frontend and backend is the instruction issue buffer. Instructions are issued from the issue buffer as their data dependencies are resolved, without regard to their original program order. In Figure 4 there is a single, unified issue buffer that services all the execution units. In most current designs, the issue buffer is partitioned according to classes of execution units. For example, all the memory instructions may be handled by one issue buffer, all the integer arithmetic instructions may be handled by another, and all the floating point instructions may be handled by a third. The maximum issue width is I. In many designs $I \geq D$, but I and D may be equal, or D could even be larger than I. In current superscalar processors, I and D are approximately four, with F being somewhat larger -- perhaps as large as eight.

Instructions issue to an execution unit or to the data cache. There may be more than one cache port or execution unit of a given type. The numbers of units and ports should balance the dynamic mix of instructions that use them. If during any clock cycle there are more ready instructions requiring a given type of unit, then an arbiter determines which instructions should be allowed to issue. Normally, this arbitration gives priority to the oldest ready instruction in the issue buffer. When instructions complete execution, they write results to the physical register file, possibly forwarding results to dependent instructions in the instruction buffer, and they report any exception (trap) conditions to their slot in the ROB.

The ROB is illustrated in Figure 5. It is managed as a FIFO buffer and is typically implemented as a small memory and two circulating pointers, a head pointer and a tail pointer. There is a ROB entry for each instruction, and the entries are arranged in the original (architected) program order. A ROB entry contains, among other things: the instruction's program counter, any exception conditions that may have occurred during its execution (shown as a bit vector in Figure 5), and an indicator of whether it is a store instruction. It also contains register state information, depending on the form of register renaming used in the processor; this is shown in the figure and will be explained later. Finally, there is a flag indicating whether the instruction has completed execution.

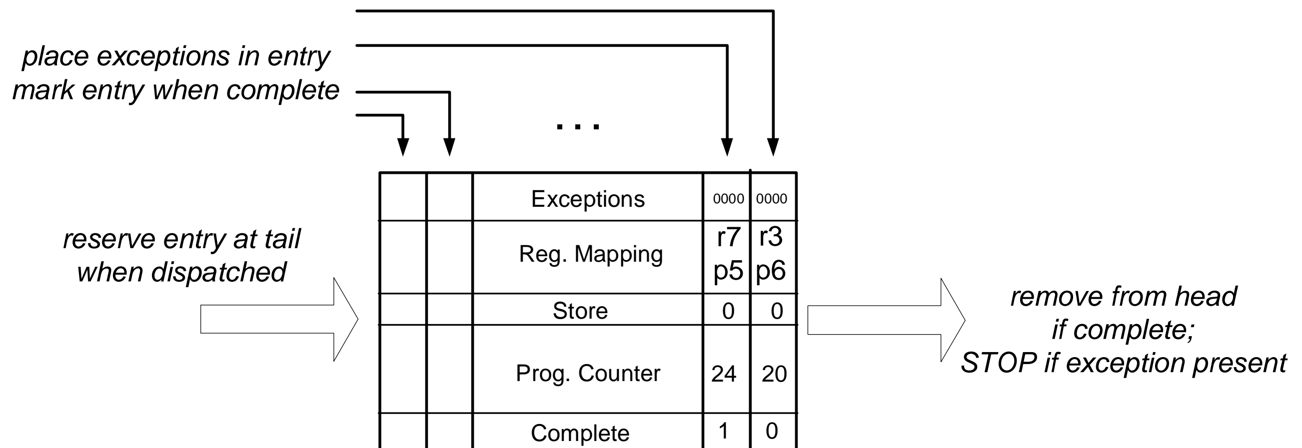


Figure 5. Reorder buffer; instructions are dispatched into the tail and exit from the head only after they have completed.

Instructions are removed from the ROB in FIFO order, but only after they have been marked as complete. At that point the instruction is *retired*, and any changes it makes to architected state are *committed*. This in-order commit assures that changes to observable state occur in the original program order. When an instruction commits its state, its result register (if any) becomes part of the architected state

© 2007 by J. E. Smith

and any memory store values are committed by writing them into the data cache; the Store flag produces a signal to the load/store unit indicating that a store to memory (or to cache memory) should take place. Register state is implicitly committed by discarding previous register mapping information, thereby making it impossible to back-up the architected state beyond this point; this is explained below. A committing instruction that has an exception condition will cause a trap when it reaches the head of the ROB. Because instructions commit their state in-order, at the time a trap occurs the state can be restored to the precise state according the architected program counter that points to the trapping instruction. In Figure 4, the maximum retire or commit rate is *R* instructions per cycle; typically the retire rate is the same as the dispatch rate.

Before describing the cache memory and load/store buffering, we will go through a detailed example of a superscalar processor in operation.

3.2.3 Detailed Example

In this example, a sequence of instructions is executed as it would be in a superscalar processor as just described. On the left side of Figure 6 is a loop that loads values from two arrays held memory, multiplies them together, then stores the result to a third memory array. Following branch prediction (which is assumed to predict the loop terminating branch correctly) and instruction fetching, a section of the dynamic instruction sequence as it fed up the pipeline is on the right of the figure. The instructions as they are stored in memory are referred to as *static* instructions; the stream of instructions as fetched are referred to as the *dynamic* instruction stream.

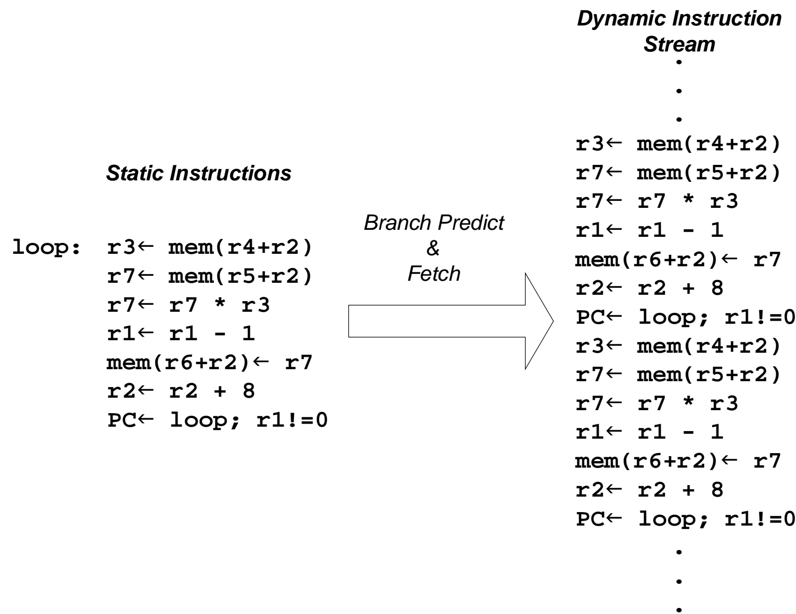


Figure 6. A block of instructions (on the left) are fetched (with the benefit of branch prediction) to form the dynamic instruction stream shown at right. A branch instruction appears as an assignment to the program counter (PC).

After being fetched, instructions are decoded and renamed. In practice, there are a number of ways that register renaming has been implemented. Here, we discuss one way that is commonly used and is conceptually simple (at least when compared with the other methods). The register renaming process is illustrated in Figure 7. With register renaming, there are more physical registers than logical (architected) registers. The logical registers are assigned to physical registers ("renamed") so that only true data dependences remain; that is, false dependences involving the overwriting of result registers are

removed by providing each instruction result with an unused physical register, i.e., one that is not being used by any currently active instruction. Because logical registers are assigned to new physical registers, there is a Register Map table that keeps track of the mapping at any point in time. When the example instruction sequence is renamed, the contents of the Register Map are as shown in the Figure 7.

The first step of renaming is the assignment of source registers to the corresponding physical registers. This is done in Figure 7a. The two source registers r4 and r2 correspond to the physical registers p1 and p4, respectively. Next, the destination register is renamed. At any given time there are a number of unassigned physical registers (or if there are none available, renaming stalls). These free physical registers are contained in a free pool of registers as shown in Figure 7b. The first register in the free pool, in this case, p8, is assigned to the instruction's result register, r3. The first instruction's registers are now renamed; to complete the process, p8 is removed from the free pool and the Register Map is updated to show that r3 now maps to p8. A following instruction that reads the value from source register r3 will have its source register renamed to p8 so that it will receive the value produced by the first instruction. The left column of Figure 8 shows the first three iterations of the example loop in renamed form.

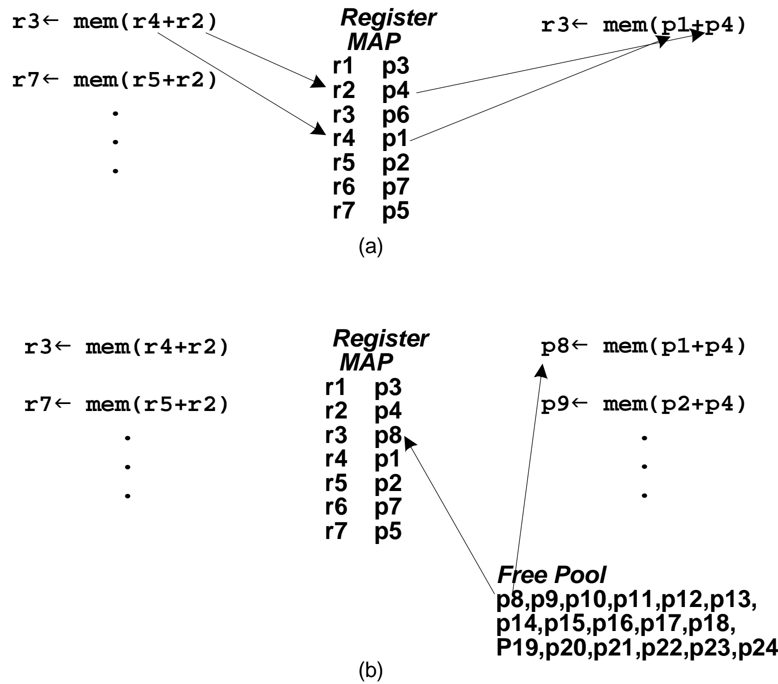


Figure 7. The register renaming process a) first source registers access the logical-to-physical register map to find their current mappings b) then the first physical register in the free pool is assigned to the result register and the register map table is updated.

After instructions are renamed and dispatched into the issue buffer and reorder buffer, the in-order front-end of the superscalar processor has done its job. Next, the out-of-order backend issues, executes, and commits the instructions. In this example, we assume a 4-wide superscalar processor where the dispatch, issue, and commit widths are all four (refer back to Figure 4). We also assume that there is a single load/store unit; i.e., there is a single data cache port. There are two integer ALUs and a single multiplier. The latency of the data cache is 3 cycles, the integer ALU is 1 cycle, and the integer multiplier is 5 cycles.

© 2007 by J. E. Smith

The timing for the overall process is shown in Figure 8. The figure shows the cycles when dispatch, issue, completion of execution, and commit take place for each instruction. First, we observe that the instructions are dispatched, in order, at the rate of four per cycle. Instructions can issue out-of-order as soon as their operands are ready; we assume that at least one cycle must lapse between dispatch and issue. During cycle 1, the first and fourth instructions issue. This assumes they have no outstanding data dependences at the time the example starts. The second and third instructions do not issue. The second cannot issue because it is a load/store instruction and there is only one load/store unit. The third cannot issue because it depends on the first two instructions, and they have not completed. In the second cycle, the second, seventh, and eighth instructions issue, again illustrating out-of-order issue. In this instance, instructions from two different loop iterations issue at the same time.

Renamed Stream	dispatch	issue	complete	commit
p8 ← mem(p1+p4)	0	1	4	5
p9 ← mem(p2+p4)	0	2	5	6
p10← p9 * p8	0	5	10	11
p11← p3 - 1	0	1	2	11
mem(p7+p4)←p10	1	3	12	13
p12← p4 + 8	1	2	3	13
PC ← loop; p11!=0	1	2	3	13
p13← mem(p1+p12)	2	4	7	13
p14← mem(p2+p12)	2	5	8	14
p15← p14 * p13	2	8	13	14
p16← p11 - 1	2	3	4	14
mem(p7+p12)← p15	3	6	15	15
p17← p12 + 8	3	4	5	15
PC ← loop; p16!=0	3	4	5	15
p18← mem(p1+p17)	4	7	10	15
p19← mem(p2+p17)	4	8	11	16
p20← p19 * p18	4	11	16	17
p21← p16 - 1	4	5	6	17
mem(p7+p17)← p20	5	9	18	19
p22← p17 + 8	5	6	7	19

Figure 8. Three iterations of the example instruction stream after renaming. Dispatch, issue, complete, and commit cycles illustrate out-of-order instruction issue and in-order instruction commit.

It is assumed that at least one cycle elapses between completion and commit. Consequently, the first instruction commits at cycle 5, and the second commits at cycle 6. Returning to Figure 7, note that the physical register assigned to r3 before it was renamed was p6; at the time the first instruction commits, register p6 can be returned to the free pool. Then, the third and fourth instructions commit at cycle 11. In this case, the commit of the fourth instruction is delayed until the third has completed.

This example shows only a small section of the loop's execution. In the steady state, executing a single loop iteration takes three cycles; the loop contains seven instructions, so the sustained execution rate is 2.3 instructions per cycle (IPC). In this example, the load/store unit is a bottleneck; each loop iteration contains three load/stores so this limits the rate at which the loop can be executed. If there were an additional load/store unit, then the execution rate would become 3.5 IPC; in this case, the bottleneck is the instruction fetch unit. It is assumed that the fetch unit can fetch only contiguous instructions; because the loop has seven instructions, this means that two fetch cycles are required per loop iteration: in the first cycle four instructions are fetched and in the second the remaining three are fetched. The average is 3.5 IPC.

In the example, we also observe that the instructions complete execution out-of-order. However, because it is required that precise architected state be maintained for exception conditions (traps), instruc-

© 2007 by J. E. Smith

tions commit their result in-order. With the renaming method used here, the committing of architected state means that the *previous* physical register assigned to a given architected register can be returned to the free pool; its value will no longer be needed to back up the state if there is an exception. The ROB illustrated in Figure 5 shows the previous register mappings for the first two instructions in our example sequence. This information is used to back up the state when an exception is detected. Instruction commit times are shown in the right column of Figure 8.

Figure 9 illustrates the state recovery and back-up that occurs when an exception is detected. The Figure shows the ROB at cycle 16 (refer Figure 8) when the load to register 19 is at the head of the ROB. This load instruction happens to be at PC x5C. We assume that this load instruction has encountered an addressing exception, so one of its exception bits is set. The current register map (rename table) is on the left. When the ROB mechanism detects the exception, it stops committing instructions. It then goes through the ROB from tail to head, restoring register mappings; the ROB entry contains the previous mapping at the time this instruction was renamed. Then, after this register back-up process is done, the program counter is loaded with the address of the faulting instruction (held in the ROB). Finally, the store queue is flushed. This means that the store instruction following the load will not take effect. At this point, the architected state is exactly the same as if instructions were executed and completed in order, and the load instruction faulted.

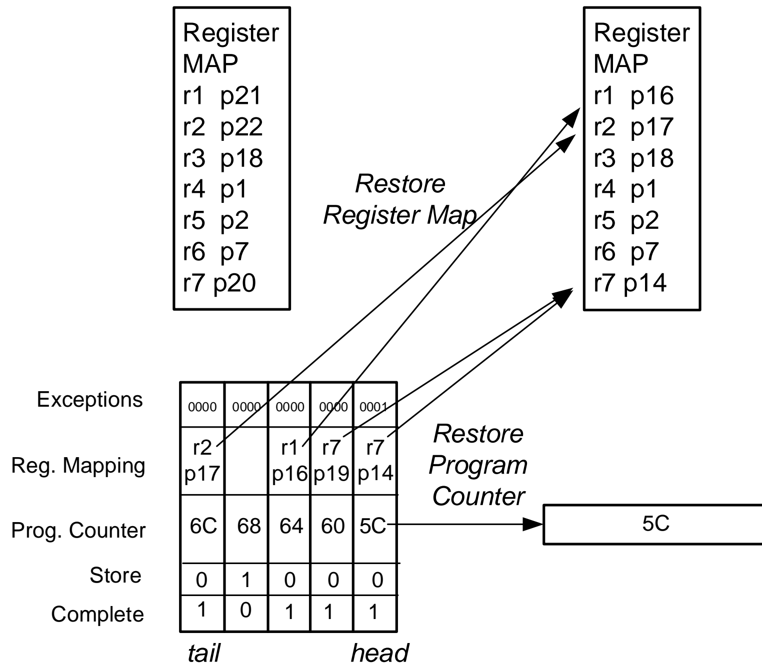


Figure 9. Example of ROB restoring architected state after an exception. The instruction at the head of the ROB has an exception. The register mapping and PC are backed up, and the pending store instruction is flushed.

3.2.4 Cache Memory and Load/Store Buffering

Of particular interest in multiprocessor system implementations is the memory path consisting of the cache memory (the L1 data cache in particular) and associated load/store buffering. It is through loads and stores that a processor core interacts with other processors in the system. This interaction is important for performance reasons, as cooperating threads communicate data through memory, but it is also important for architecture reasons as we saw in Chapter 2, where the ordering of memory accesses among multiple processes must be carefully managed and where memory must be kept coherent.

© 2007 by J. E. Smith

A significant problem that must be solved in the memory path is the resolution of data dependences involving memory data. Register renaming partially solves the data dependence problem for register instructions, and renaming is done while instructions are still in-order, before instruction issue. The remaining register dependences, the true data dependences, are resolved at issue time by comparing operand register identifiers. With memory operations, however, exact memory addresses are not known until *after* a memory load or store issues and its address has been computed. This late (post-issue) determination of memory addresses makes it significantly more difficult to issue loads and store out-of-order.

From a performance perspective, it is desirable to execute loads and stores out-of-order, just as for register instructions. Because loads are normally on the critical path for computation and stores are not, there are significant advantages when loads can pass stores that appear earlier in the instruction stream. There are also some performance advantages to letting loads issue out-of-order with respect to other loads, depending on availability of address registers. Load/store reordering takes place in the example given earlier in Figure 6; the advantages can be seen in timing data shown in Figure 8 where loads passing stores is a major contributor to performance gains (especially note the out-of-order instruction completion times). In the example microarchitecture it is assumed that stores can issue and compute their addresses as soon as the address operands are available. The data operand for a store does not have to be ready in order for the instruction to issue; it can be forwarded to the load/store unit when it becomes available.

Figure 10 illustrates a typical, aggressive implementation that allows load/store reordering. It is easiest to first describe the handling of typical load instructions, then store instructions, and finally their interactions. Along the top of the figure is the main path through the data cache. Load instructions are issued, their addresses are computed, and the data cache is accessed in parallel with the translation lookaside buffer (TLB). The TLB translates the upper (page number) bits of the address while the cache accesses (both data and tags) use lower, untranslated address bits. The translated address is then compared with the cache tags to determine if there is a hit. If so, data is returned to the load's result register (and is possibly forwarded to dependent instructions waiting in the issue buffer). If there is a cache miss, then a request is made to main memory, or the next level in the cache hierarchy, the L2 cache. Information related to a load instruction that misses is placed in a Miss Status Holding Register (MSHR); this information is used when the line is returned from the memory system. It indicates where to place the requested line in the cache and routes the data to the proper register.

For store instructions, there is a store queue, shown near the bottom of Figure 10. At the time store instructions are dispatched into the issue buffer (while instructions are still in order), they are assigned an entry in the store queue. One of the functions of the store queue is to hold stores in their original program order so that their results can be committed in the architected program sequence. This allows a precise architected state to be constructed if an exception condition occurs. A store instruction also sets a flag in the ROB as described above. When the store's entry is at the head of the ROB, and if it is exception free, it sends a *store commit* signal to the store queue. Then, the store is committed by writing its result to the cache.

In practice, the store data does not have to be written to the cache immediately. Because stores are less critical to performance than loads, they may be kept temporarily in a store buffer as shown in the lower right corner of the figure. When an otherwise unused cache time slot becomes available (or the store buffer is full), the contents of the store buffer are written to the cache. In many implementations, this store buffer allows the *coalescing* of writes to the same cache line. It is often the case that there are multiple stores to the same cache line that occur close together in time. The coalescing store buffer com-

bines these so that when the cache is written, the values of the combined stores can be written at the same time.

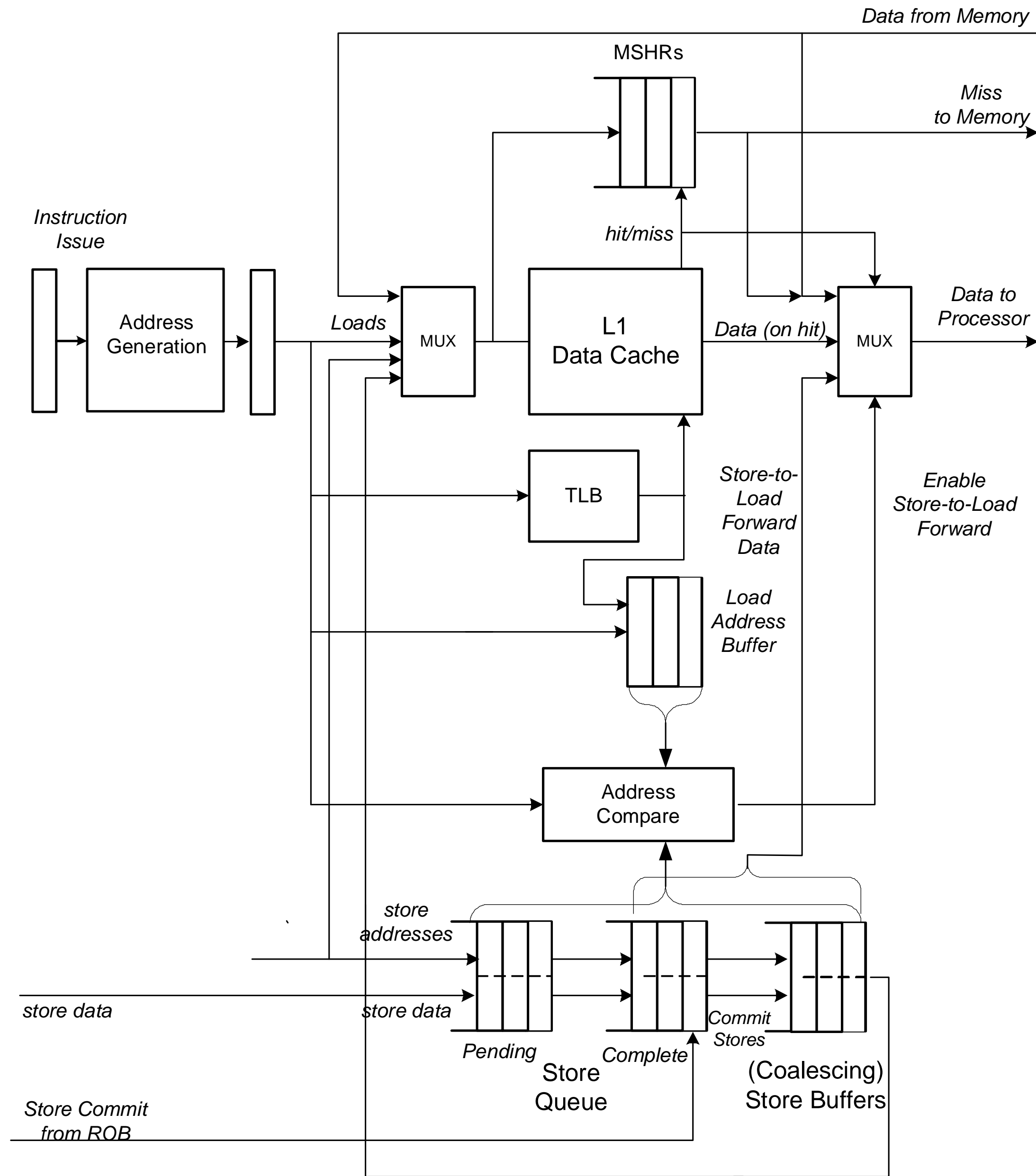


Figure 10. L1 data cache and buffering subsystem that allow load/store reordering with forwarding of load data.

For resolving memory data dependences involving loads and stores, it is the store address that is important, not the store data. Hence, a store instruction is allowed to issue as soon as its address registers are ready; the data register does not necessarily have to be ready. When a store issues, its address is computed and placed into its pre-assigned slot in the store queue. If its data is also available at the time the store issues, the data is also placed into the store queue slot. If the data is not ready, it is forwarded to the store queue later, at the time it does become available. In Figure 10, the store queue is conceptually divided into two sections. The pending section holds stores whose data is not yet available, and the complete section holds stores whose address and data are both known.

In this implementation, load and store instructions issue in their architected program order, so that the stores fill in the store queue in program order. When a load issues, its address is checked against the stores in the store queue to see if its address matches a store address in the queue. There are three possible actions that may result from this comparison. 1) If the load's address does not match the address of any of the stores, nothing more needs to be done; the normal load path described above is followed. 2) If the load's address matches a store address in the complete section of the store queue, then the store data can be forwarded immediately to the load. This is done via the multiplexer (MUX) at the upper right of the figure. The forwarded data replaces the data coming from the data cache. 3) If the load address matches a store address in the pending section of the store queue, then there is a store-to-load data dependence, but the data is not yet available. In this case, the load instruction is placed in the load address buffer. It waits there until the data for the matching store becomes available, and at that time the data is forwarded to the load instruction.

A more detailed drawing of the buffering and comparison logic is in Figure 11. This approach allows loads to pass store instructions if there are no address conflicts, indicating memory data dependences. To re-iterate, if a load address does match a queued store address, then the data value is forwarded from the store to the load as soon as it is available. If the store data is already available in the store queue (or the coalescing store buffer – not shown) then it is immediately forwarded (Forward 2 in the figure). If the data is not available, then the store queue entry that has the matching address (SQ tag) is recorded in the load address queue. Then, when the store data arrives, its store queue entry value is compared with the SQ tags. A match causes the store data to be forwarded to the load (Forward 1 in the figure). This approach just described is the one used in the above superscalar processing example (see Figure 8).

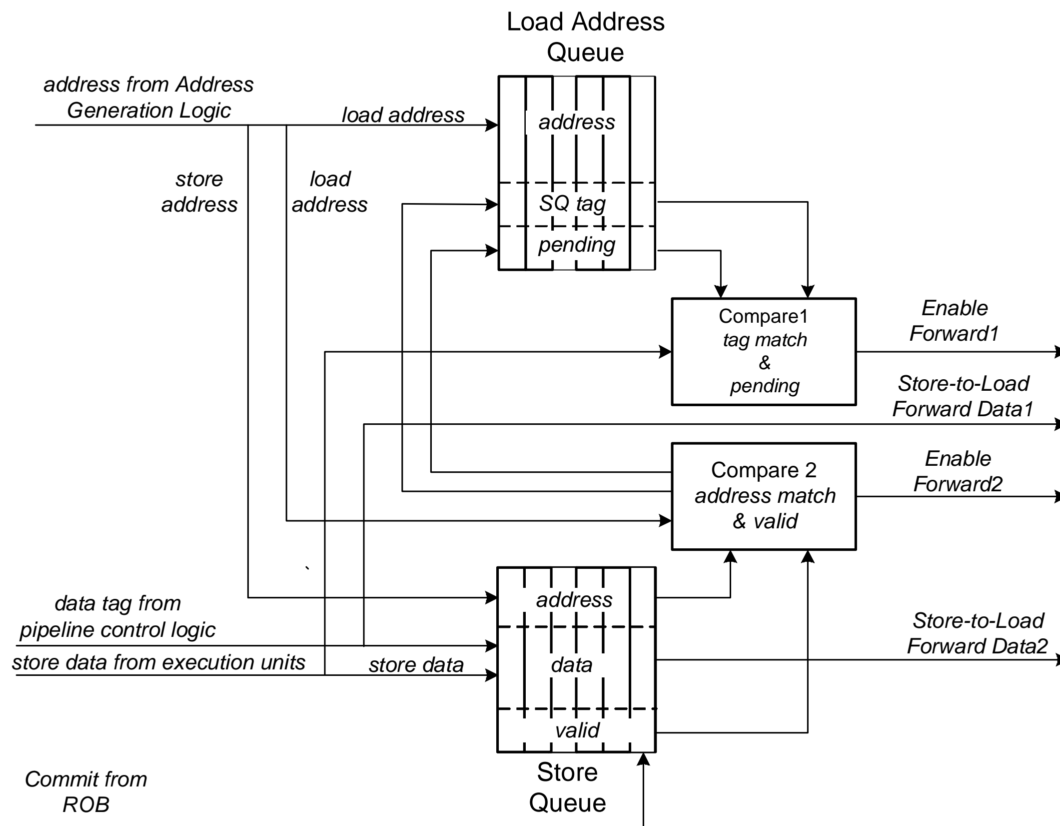


Figure 11. Detailed drawing of load/store buffering and comparison logic.

An even more aggressive approach allows load instructions to issue out-of-order with respect to store instructions, and with respect to each other. See Figure 12. This figure shows additional buffering and associated logic that is added to the logic shown in Figure 11. With this approach, address conflicts indicating memory data dependences are not known at the time a load issues; it is essentially predicted that there will be no address match (and no dependence) with preceding store instructions. This is often the case, but it is not always the case. Consequently, load instruction addresses are buffered after they access the data cache (this is the right half of the Load Address Queue). Then, all store instructions that logically precede the buffered loads compare their addresses with the completed load addresses when the store addresses become available. If there is a match, and the load instruction did not get data forwarded from this store, then the load instruction got the wrong data (and all following instructions – or at least following dependent instructions) must be flushed and re-executed. This backup and recovery can be done via the same reorder buffer mechanism as for precise traps.

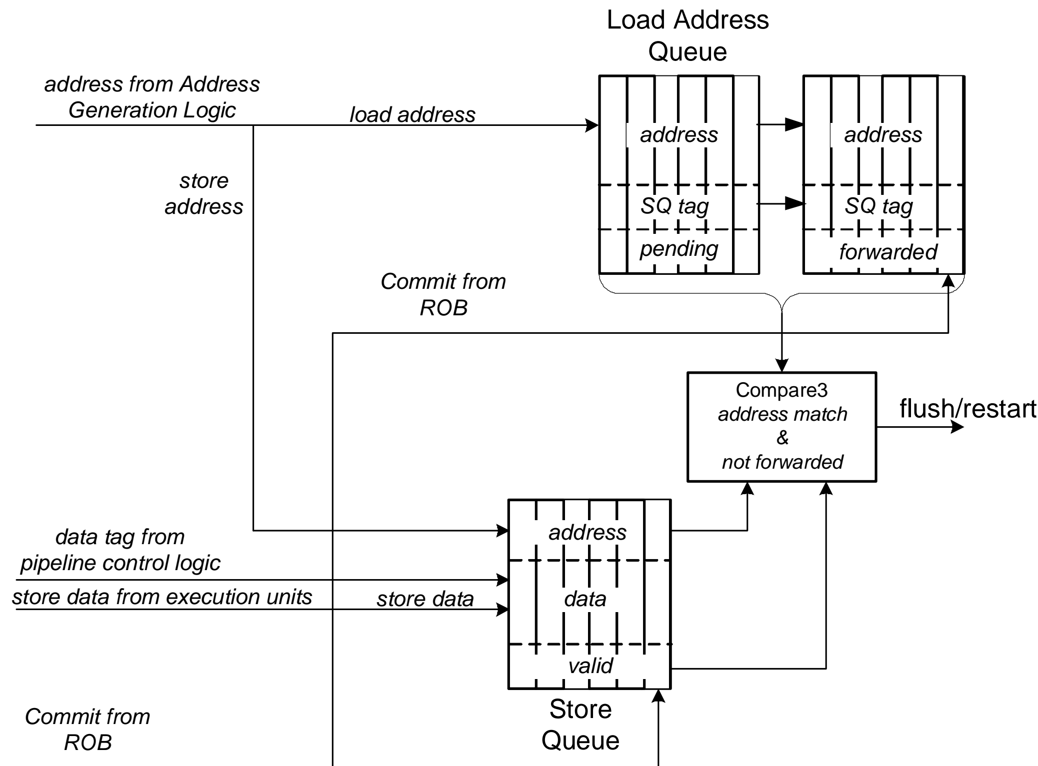


Figure 12. Portion of load/store unit that implements speculative issuing of load instructions before prior store addresses are known.

Of course, the full pipeline flush can degrade performance, depending on how often it occurs. Consequently, there have also been a number of proposals to avoid a complete pipeline flush. One approach is to selectively flush only the load instructions and following instructions that depend on its result. Another approach is to track the history of past address conflicts (and flushes) to predict future ones. If a future one is predicted by the tracking mechanism, then the load instruction is delayed from issuing until the store address is known [32]. For example, this is the approach taken in recent Intel microprocessor designs [33].

3.2.5 Balanced Superscalar Processor Designs

A superscalar processor is composed of a number of interacting hardware resources, which include the various components that perform instruction fetch, decode, rename, issue, execution, and commit.

© 2007 by J. E. Smith

For example, the issue buffer is a resource, the integer adder is another. As instructions pass through the processor they use some of the resources (e.g., issue buffer slots), and are passed from one resource to another through interconnection structures (often point-to-point busses). For optimal performance, all of these resources should be balanced, that is, they should be in proper proportions so that bottlenecks do not result in inefficiencies. In the example of Figure 8, we saw such a bottleneck. For the code shown, the load/store unit was a bottleneck that limited IPC to 2.3. This meant that some of the other resources in the processor were used inefficiently; for example, the rename unit which is capable of renaming four instructions per cycle was under-utilized. Of course, there will always be some bottleneck, but in a balanced design, most of the resources should be near their limit for most of the code sequences that are encountered. For a better balanced design (at least for the example code), one could use two load/store units in the example microprocessor; or, alternatively, one might decide to design a 3-wide superscalar processor and save resources in other parts of the processor.

When processor resources are shared, as they are when processor multi-threading is used, the matter of resource balance becomes an important one. We consider processor balance in the remainder of this section. To guide the discussion, we show a diagram of the major superscalar processor resources and the relationships that hold in a balanced design. We will discuss a number of these relationships.

First, consider the relationship between instruction issue width and the ROB and issue buffer sizes. Referring back to our performance discussion in Chapter 1, issue width is a bandwidth resource and the ROB and issue buffer are capacity resources. We begin with the ROB, which defines the window over which ILP is extracted. There have been a number of studies that have investigated the relationship between the sustainable issue width I and window size W under ideal conditions (no cache misses or branch mispredictions). It is often approximated as a quadratic relationship: $W \approx I^2$ or $I \approx W^{1/2}$. This is only a rough approximation, and the real relationship is very much dependent on the program being executed. For example, a study by Eyerman et al. [20] show that for a number of benchmarks, the relationship is between quadratic and quartic : $W^{1/4} \leq I \leq W^{1/2}$. Furthermore, this assumes that all instructions have a single cycle latency. If the average instruction latency is l, then the relationship becomes $l \cdot W^{1/4} \leq I \leq l \cdot W^{1/2}$.

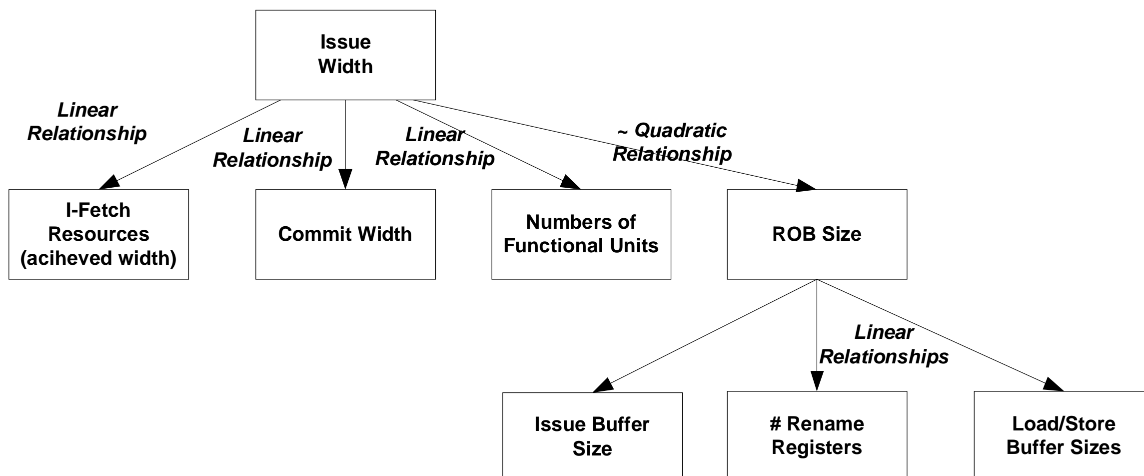


Figure 13. Relationships between microarchitecture structures in a balanced superscalar design.

Table 1 shows the issue widths and window (ROB) sizes for a number of superscalar processors, and this approximately quadratic to quartic relationship is evident in the fifth column. As noted earlier in

© 2007 by J. E. Smith

this chapter, the issue buffer is essentially an engineering optimization. It holds only the un-issued instructions from the window; these are the only ones that need to consume expensive (in both hardware and power) instruction issue resources. Referring again to Table 1, we see an approximately linear relationship between the ROB size and the issue buffers size. Generally speaking, the issue buffer is about 1/3 the size of the ROB.

Table 1. The relationship between window size (ROB) and issue width for some real processors.

Processor	Reorder Buffer Size	Issue Buffer Size	Issue Width	$\log_2$(ROB) ÷ $\log_2$(Issue Width)	Issue Buffer Size ÷ROB Size
Intel Core	96	32	4	3.3	.3
IBM Power4	100	36	5	2.9	.4
MIPS R10000	64	20	4	3.0	.3
Intel PentiumPro	40	20	3	3.4	.5
Alpha 21264	80	20	4	3.2	.25
AMD Opteron	72	24	4	3.1	.3
HP PA-8000	56		4	2.9	
Intel Pentium 4	120		3	4.4	

Next, we consider some of the other superscalar processor resources (Figure 13). In a balanced system, some of these scale linearly with the issue width, and others scale linearly with the ROB size. The ones that scale linearly tend to be resources in the pipeline front-end, as well as those that are bandwidth-related. The resources that scale quadratically tend to be the back-end buffer resources, including the ROB, issue buffer, load/store buffers, and physical (rename) registers. The I-fetch and commit resources scale linearly with the issue width. These resources supply the input and output of the overall pipeline and should be approximately the same as the issue width to provide a smooth pipeline flow.

The numbers of execution units are related linearly to the issue width and the instruction mix. For example, if the issue width is four and .4 of all instructions are load/stores, then there should be 1.6 load store units for balance. Of course, one can't build .6 of a load/store unit, so a balanced design should have either 1 or 2 units. If there is only one unit (as in our earlier example), then the load/store unit is likely to be a bottleneck for a number of programs (and this would argue for a 3-wide processor as noted earlier). It is an engineering decision whether to round up or round down when balancing functional units with issue width. In some cases the decision is simplified because the balanced number of units is one, so rounding up is the only choice; an integer multiplier is a typical example.

The quadratic-scaled resources tend to be fairly complex and consume significant power. They are composed of relatively large amounts of comparison logic that are active most, if not all of the time. This leads to the chip floor plans as derived by Olukotun et al. [34] and given in Chapter 1.

3.3 Multi-Threaded Processors

As illustrated in the timing diagram, Figure 7 of Chapter 1, many parts of a computer are used intermittently, including the processor, itself. A superscalar processor doesn't normally stall for ordinary register data dependences because of its out-of-order issue capability, but it does stall when there is a miss

© 2007 by J. E. Smith

event. And if a cache miss goes all the way to main memory, the stall can be quite long, perhaps hundreds of cycles. An in-order processor stalls more frequently, because even individual data dependences in register instructions occasionally cause stalls (See Figure 2). Whenever a processor stalls, many of its resources go unused.

By using multithreading, where a single processor supports multiple concurrent threads, hardware resources (both inside and outside the processor) can be more efficiently used. In order to support multiple execution threads, the processor maintains multiple sets of architected state – program counters and register states.

3.3.1 Early Multi-Threaded Processors

It is useful to describe the concepts of processor multi-threading by considering the way they were developed historically, particularly because there were two very notable designs in the evolution of multi-threaded processors.

THE 6600 BARREL AND SLOT

As a way of using processor hardware resources more efficiently, designers in the early 1960s came up with the idea of supporting multiple execution threads with the same set of hardware resources, including the ALU and memory data path. Perhaps the first computer system of this type was part of the CDC 6600 computer system [35]. The CDC 6600 was developed before cache memories had been invented (and at a time when main memory latencies, as measured in processor clock cycles, were much lower than today). Nevertheless, a main memory access could take ten or more clock cycles, and a processor could wait several cycles for data after issuing a load instruction.

The CDC 6600 contained a central processor (for which it is most famous), but the system also contained a set of ten peripheral processors (PPs) which were essentially small, 15-bit general purpose computers that sat on the periphery of the 6600 central processor. Although they handled I/O functions, the PPs were also able to execute portions of the OS code. Each of the PPs had a program counter and a small set of registers; however, all the PPs shared a single hardware ALU and path to main memory.

The PP's operation was described by using the metaphor of a "barrel" and a "slot". Refer to Figure 14. The ten sets of architected state were arranged around a rotating *barrel*. The barrel made one full rotation every ten central processor clock cycles (not coincidentally the same as the main memory latency). At any given cycle, the state belonging to one of the PP's was exposed to a *slot* which was the pathway to the shared hardware. During that one cycle, the PP in the slot could execute one instruction before the barrel rotated so that the next PP's state was in the slot. Now, because the memory latency matched the full rotation time of ten cycles, if a PP performed a load instruction, then the data would be ready for use the next time the PP had access to the slot. From an individual PP's viewpoint, all instructions, including memory instructions, executed in a single cycle, where its cycle was ten times the central processor cycle. By using the barrel and slot, all ten PPs were implemented with only one ALU, one memory path, as well as other associated data paths and buses.

© 2007 by J. E. Smith

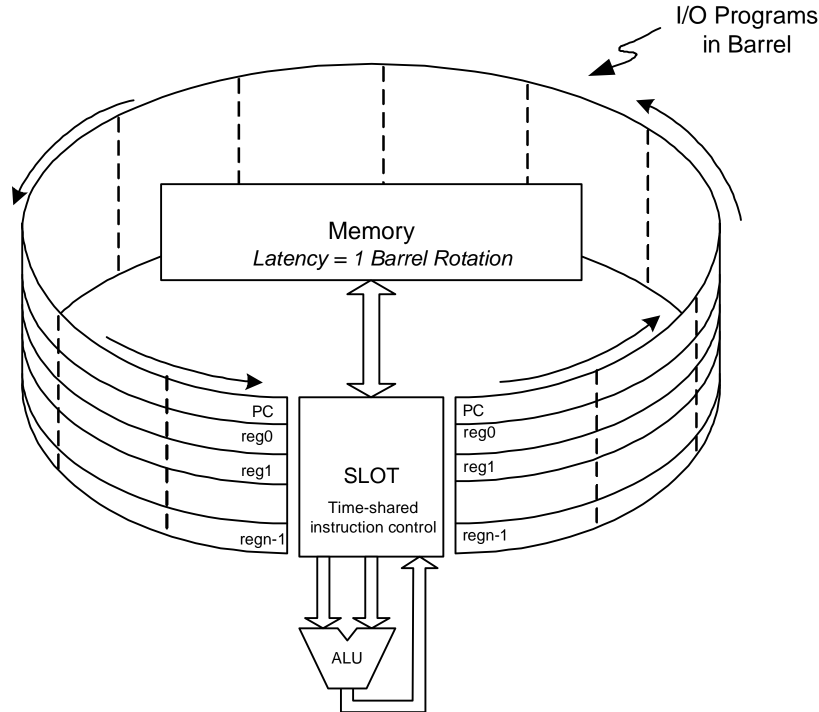


Figure 14. CDC 6600 Barrel and Slot multi-threading.

DENELCOR HEP

The second notable computer that used multi-threading was the Denelcor HEP [3]. The HEP was a general purpose computer, targeted at numerical applications. It had multiple processors, each of which was multi-threaded. A single multi-threaded processor is illustrated in Figure 15. It used an in-order pipeline that could issue one instruction at a time. The HEP maintained multiple register sets, held in a common Register Memory. For each register set there was a program status word (PSW) that contained the program counter and other state such as trap conditions.

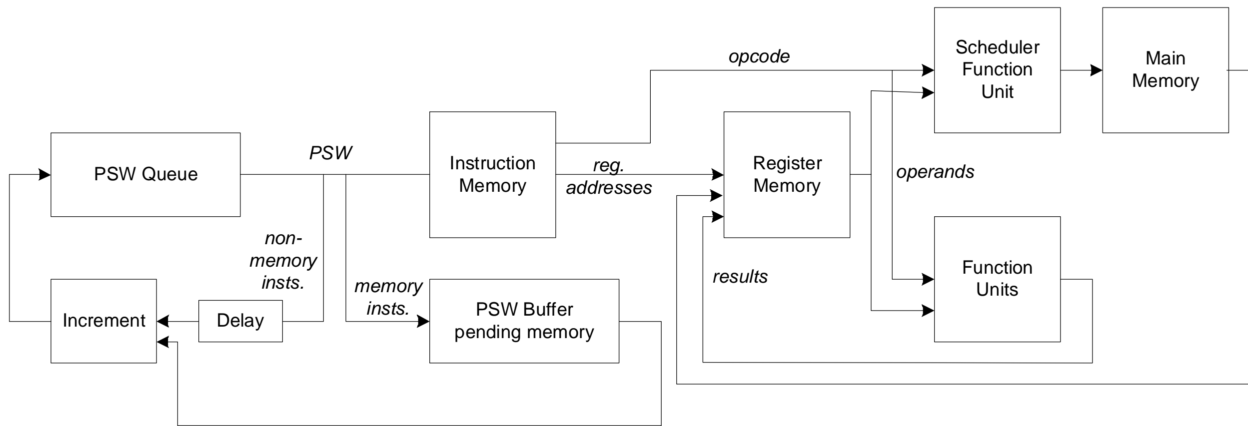


Figure 15. Block diagram of the Denelcor HEP.

Referring to Figure 15, the PSW queue is a hardware-managed run queue that holds the current PSW value for all the threads that are ready to execute. Each cycle, the PSW at the head of the queue is selected, and it begins flowing down the pipeline where it reads the next instruction from an instruction memory (cache), accesses registers, then executes the instruction. The PSWs for instructions that per-

© 2007 by J. E. Smith

form a non-memory function, say an add or a multiply, are delayed a length of time that is at least as long as the instruction's execution time. The PC in the PSW is then incremented and the PSW is placed back into the PSW queue. Memory instructions are handled by a scheduler functional unit that makes memory requests and receives data back from memory. The memory system consists of an interconnection network that combines a number of multi-threaded processors with main memory and I/O devices. A memory access has a fairly long, variable latency, so PSWs for memory instructions are removed from circulation and placed in a holding buffer, pending the completion of the memory operation. When the memory operation is complete, the PC in the PSW is incremented and the PSW is placed back into the PSW queue.

A single HEP processor could support 120 threads. The delay mechanism in the PSW loop simplifies the pipeline hardware because whenever a PSW is at the head of the queue, all of its operand registers are guaranteed to be ready. Hence, there are no pipeline instruction stalls for register dependences; and, because the delay is at least as long as the execution time, there is no need for forwarding logic. The main computation pipeline is eight stages deep. So, if there are at least eight active threads circulating in the main PSW loop, the processor can run at full efficiency without the complication of stall and forwarding logic.

3.3.2 Multi-threaded Superscalar Processors

The two example processors given in the previous subsection were either sequential (CDC6600 PPs) or used in-order pipelines (HEP). Most processors today, however, are superscalar out-of-order processors, and if multi-threading is implemented in a superscalar processor, then a number of interesting issues and design alternatives arise. Because each pipeline stage in a superscalar processor can handle more than one instruction per cycle, bandwidth resources can be assigned to different threads, even within the same cycle. There also are more sharable capacity resources in a superscalar processor, for example issue buffers, the ROB, load/store buffers, etc.

To help introduce some of the issues, we again use an example -- this time a multi-threaded superscalar processor. Figure 16 illustrates multi-threading as implemented in the Intel Pentium 4 microarchitecture. Intel literature refers to it as *hyper-threading,* but it is the same thing as multi-threading (See Marr et al. [6]). The Pentium 4 can support two threads simultaneously; these are represented in Figure 16 with the colors red and yellow.

There are two separate program counters; these are part of the architected state and identify the two execution threads. The trace cache is an advanced instruction cache that holds dynamic sequences of pre-decoded instructions [36]. The dynamic instruction sequences in a trace cache follow predicted branch paths. Only one thread can access the trace cache in a given cycle, so the two threads fetch on alternate cycles whenever both have a valid request; otherwise one thread can fetch every cycle. This is essentially a version of a *round-robin* scheduling policy as applied to two threads. Allowing both threads to fetch simultaneously in the same cycle would require a two-ported trace cache, a significant complication, and it would probably not improve the efficiency of trace cache very much.

© 2007 by J. E. Smith

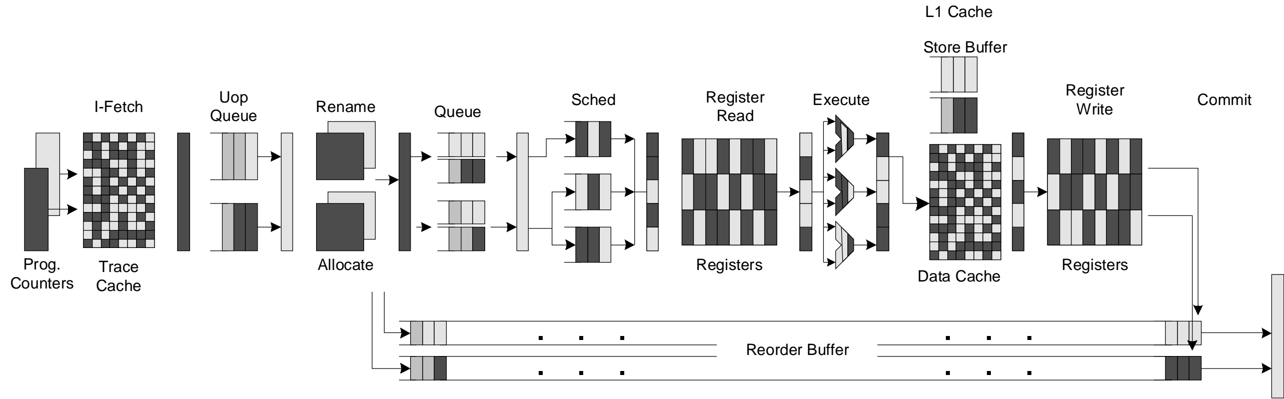


Figure 16. Intel Pentium 4 hyperthreading.

The trace cache is a capacity resource and is shared among the two threads, using an LRU replacement algorithm across the two threads. As with many other shared capacity resources, the trace cache entries are tagged with a thread identifier (TId) to separate the entries belonging to different threads. This is illustrated in Figure 17. In a cache memory or a shared buffer, all the entries are tagged with a TId. Similarly, the TId of the currently active thread is appended to the address being used to access the cache or buffer. For example, in Figure 17, the TIds are compared along with the conventional tags to decide if there is cache hit or miss.

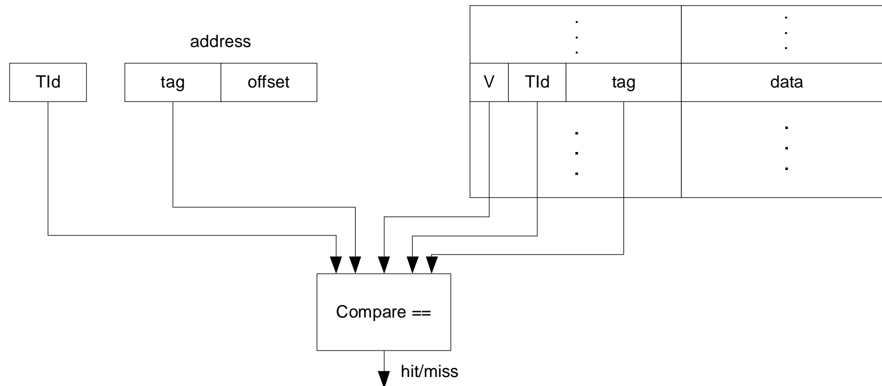


Figure 17. A thread identifier (TId) separates the entries belonging to different threads in a shared buffer or memory.

The uop (micro-operation) queues separating the I-fetch and rename stages are partitioned (each thread has its own queue). This assures that a thread can make independent forward progress when the other is stalled. As with the I-fetch stage, uops from the two threads are processed in the register rename stage on alternating cycles (unless only one thread is active). The two threads have separate rename and allocate logic. The rename logic holds the logical-to-physical map table for a given thread. The allocate logic assigns buffer entries to a given thread; these include the ROB and load/store buffers. In conjunction with the rename logic, the allocator also assigns physical registers. The physical registers are managed as a shared pool using thread identifiers. After register renaming, the uops are placed into two sets of queues, one for memory operations and one for non-memory operations; these queues are also partitioned between the threads.

© 2007 by J. E. Smith

Then five instruction schedulers (only three are shown in the figure) decide which instructions are ready to issue to the available functional units and memory ports (Intel sometimes calls this "dispatch"). The scheduling logic makes its issue decisions regardless of the thread; that is, unlike the previous pipeline stages it does not alternate threads on a clock cycle basis, but mixes operations from the threads within the same clock cycle. However, to assure some fairness between the threads there is a limit to the number of uops a thread can have in each of the scheduling queues (the limit depends on the queue's overall size).

Because instructions issue from both threads in a mixed manner, the execution units also process the operations from the two threads in a mixed manner. Instructions write registers when they complete. The commit stage, as with earlier pipeline stages, retires instructions from the two threads on an alternating (round-robin) basis.

In summary, multi-threading is implemented with a combination of partitioned and shared (pooled) capacity resources, and by a combination of alternating per clock cycle and mixed clock cycle allocation of bandwidth resources. The approach to resource partitioning and sharing is the result of a number of design decisions regarding performance and fairness goals. In the next section we will discuss shared resource management in more detail, with the goal of shedding light on the way that both bandwidth and capacity resources are allocated.

3.4 Shared Resource Management

A multi-threaded processor, especially a superscalar processor, is composed of a number of hardware resources that are shared among threads. This is not unlike the situation that occurs in conventional computer systems where resources are shared among threads under OS management. In a conventional computer system, the shared hardware resources are the processor, main memory, and I/O, and the OS allocates these resources among the threads or processes. For example, the OS uses page replacement algorithms to manage the main memory resource, giving certain amounts of physical memory to the running processes. The OS also decides which process should be allowed to use a processor during a given time interval and invokes context switches to manage the processor resource accordingly.

The situation with resource management in a multi-threaded processor is similar, except that many of the microarchitecture-level resources are not visible to system software, so they cannot be managed by a conventional OS. Furthermore, they typically need to be managed on a much smaller time scale than the OS would be capable of. Finally, there are a lot of implementation-specific resources in a multi-threaded microarchitecture (consider the Intel hyper-threading example); the resources aren't as generic as "processor" and "memory". For all of the reasons just given, the allocation of resources in a multi-threaded processor falls on the hardware and is largely implemented by the hardware designer.

The overall management of resources in a multi-threaded processor fits the generic picture given in Figure 18. At the bottom level, the processor is a collection of bandwidth and capacity resources. This figure shows only the processor, but the full system, including the memory hierarchy, can be divided into bandwidth and capacity resources in a similar way. For example, main memory has a capacity and a bandwidth. In any event, the resources, both bandwidth and capacity, must be allocated to the threads in some way, and this is done via policies that implement overall objectives. The bandwidth resources are typically allocated on a short-term basis, perhaps on a per cycle basis (for fully pipelined resources). A capacity resource is held for some period of time (often not known in advance and its time of occupancy may have a high variance).

Referring again to Figure 18, the objectives are overall goals to be achieved. The policies are algorithmic plans of action to achieve the objective, and the policies drive the control of the actual mechanisms that implement the policies. We will briefly discuss objectives in the next subsection, and then describe mechanisms and policies in subsequent subsections.

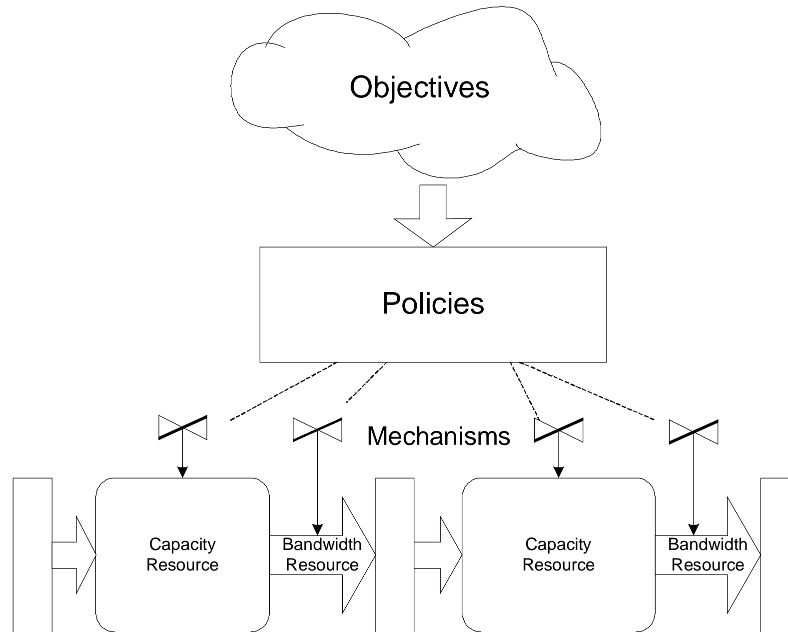


Figure 18. Objectives, policies, and mechanisms.

3.4.1 Objectives

The objectives or overall goals generally involve some measure of performance, fairness, and/or isolation. In some cases the objectives may be only qualitatively describe the goal(s) and/or the policies and mechanisms provide only a rough approximation to the objective. In other words, this is not a completely formal process. Often qualitative objectives are tested via heuristics and simulation; for example, a number of specific policies/mechanisms may be implemented in a hardware simulator and tested for a number of benchmarks. The one that is judged to come closest to meeting the objectives may then be selected for hardware implementation. Following is a description of major classes of objectives.

PERFORMANCE

There is almost always some performance objective. The specifics of a performance objective often depend on the application, however. Following are some of the major performance objectives.

- Optimize aggregate performance over the set of actively running threads. Aggregate performance may be measured as the average over all the threads. Achieving this objective may often mean that the individual threads are not running at top-speed, but total performance for all the threads is maximized. This might be an objective in some server systems, for example.
- Provide assured performance level(s) for certain threads. This occurs if there are (soft) real-time requirements for some of the threads. This may happen with multimedia applications where, for example, a certain frame rate must be achieved in order to provide good quality video. This objective may also be used in combination with others. For example some thread or threads may have a real-time requirement, and any left-over resources may be used to opti-

mize aggregate performance for the other threads. A related situation occurs when there is an interactive thread where user response time is critical and other background tasks are less time critical.

FAIRNESS

The term "fairness", by itself is often not a well-defined goal; there are a number of ways that fairness can be defined. For example in the Intel description of Pentium 4 hyper-threading [6], fairness is stated as a motivation for some of the design decisions, but a quantifiable definition of fairness is not given; rather it is a rather informal objective. Sometimes what is meant by fairness could be better stated as the absence of obvious unfairness, for example if a thread's performance is seriously degraded when compared with other threads, or if the thread is completely "starved" by other threads. A round-robin policy as used in the 6600 PPs, or in some of the pipeline stages of the Pentium 4 would appear intuitively to provide some degree of fairness.

Most of the quantifiable definitions of fairness have been relative measures; performance is compared to the baseline performance when a thread is given all the processor's resources for itself. One such measure combines aggregate performance with fairness. This measure is the harmonic mean of the speedups for concurrently executing threads [27]. That is, the speedup of thread *i*, is $S_i = time_{unshared} / time_{shared}$. Note that in the situation we are discussing, the "speedups" will generally be less than one, indicating a slowdown when a thread runs concurrently with other threads, so we are trying to minimize the slowdowns. Then, the proposed performance/fairness measure, the harmonic mean, is $n \cdot \Sigma(1/S_i)$. A property of the harmonic mean is that it is consistent with Amdahl's law; as the speedup of one thread is increased significantly with respect to the others, its contribution to the harmonic mean diminishes. Consequently, optimizing harmonic mean tends to balance the performance losses (slowdowns) among the threads.

Another fairness measure based on the same speedup metric is to make all the speedups (actually slowdowns) the same for all concurrently running threads [10]. That is, all the concurrent threads should be slowed down by the same percentage when compared with the performance they would achieve when running alone with all the available resources. This notion of fairness essentially attempts to "share the pain" of multi-threading by degrading the performance of all the threads by the same percentage. One problem with this approach is determining the baseline (unshared) performance level.

Fairness is often a secondary objective. For example, if aggregate throughput is the primary objective, it may be the case that the best throughput occurs if some particularly inefficient thread never runs. This would seem to violate an intuitive notion of fairness. Hence, when optimizing aggregate performance, one might also add some fairness criteria, or define the aggregate performance measure so that some degree of fairness is maintained. This is the objective of the harmonic mean measure described above. Aggregate performance with some amount of fairness appears to be the objective of Pentium 4 hyper-threading.

Another definition of fairness may focus on resources rather than performance. That is, a thread should get its fair share of resources (a "share the resources" approach). For example, if there are *n* active threads, then each should get at least $1/n$ of each of the hardware resources [21].

ISOLATION

Because threads run concurrently and share resources, the execution of one thread may influence (interfere with) the performance of another, even if the threads are from totally independent programs. This means that the performance observed for one thread may vary considerably depending on the

© 2007 by J. E. Smith

other threads that happen to be running concurrently. This unpredictability can cause problems when attempting to satisfy real-time goals, or, it can simply be disconcerting to a user when his/her application displays significant performance differences for no obvious reason.

Consequently, performance isolation is sometimes a goal. That is, a thread should achieve a minimum, predictable level of performance regardless of the other threads that are running at the same time. This might be the case in a server system that is shared among multiple users, where each user observes the same performance regardless of what the other users are doing. The user may, in some cases observe better performance than the predictable level (if the other users are not using all the resources, for example). This objective is also related to real-time performance objectives, but in this case the goal is to provide isolated performance for all threads that a given user may decide to run; there is no specific real-time requirement for the individual threads.

IMPLEMENTING OBJECTIVES

As stated above and illustrated in Figure 18, objective(s) are implemented via a policy, which is a plan or course of action for achieving the objective. Although objectives may sometimes be qualitative (or even vague), policies have to be implemented, and therefore must be defined as algorithms. Mechanisms are the primitives that are directly controlled by the policies. Ideally, the mechanisms and policies should be separated, but this ideal cannot always be achieved. Separating mechanisms and policies tends to increase flexibility in implementing a variety of policies. To quote Wulf: "provide primitives, not solutions" [37]. In the context of multi-threaded processors, mechanisms are implemented in hardware, so their level of sophistication may be rather limited. And, because of the time scale, it is often impractical to implement policies entirely in software, so some or all of a policy implementation is in hardware as well. This means that policies and mechanisms may have rather limited flexibility, and their implementations are often inter-twined.

For examples of policies and mechanisms, consider Pentium 4 hyper-threading. At the fetch stage there is a *mechanism* that enables the selection of one (and only one) of the threads to perform an instruction fetch during a given cycle. The mechanism is essential a multiplexer. The *policy* is to select the threads in a round-robin fashion; the policy controls the select input of the multiplexer. The Pentium 4 could have also implemented a priority policy for instruction fetching that controls the same selection mechanism. Such a policy would always select the high-priority thread for fetching, and select the low priority thread only when the high priority thread does not need to fetch. In the Pentium 4, the round-robin policy is implemented in hardware. But, one could imagine the ability to select between the round-robin and priority policies based on software-settable control registers or boot-time parameters. As we will see in Section 3.5.3; this approach is taken in the IBM Power5 multi-threaded processor.

In following subsections we discuss first mechanisms, and then policies that are used (or have been proposed for) multi-threaded processors. All told, the above objectives (and various combinations), along with policies and mechanisms that implement them, define a very large solution space, not all of which has been explored. We will touch on some of the important points in the space, focusing on real designs and important research studies.

3.4.2 Mechanisms

We discuss basic mechanisms for sharing bandwidth and capacity resources, as well as other important aspects of mechanisms.

© 2007 by J. E. Smith

BANDWIDTH SHARING

A pipeline stage is an example of a bandwidth resource. It is available for assignment to a thread during a given cycle, and, if unallocated, it will go unused that cycle. In a pipelined processor many of the bandwidth resources are only consumed for a single cycle. But there may be instances of multi-cycle bandwidth resources such as a serial divider that, once allocated, may used for several cycles before it can be re-allocated.

In a superscalar processor, a given stage (bandwidth resource) may be able to support multiple operations during the same cycle. Then, the mechanism for sharing this bandwidth either assigns all of the resource to a single thread for a given cycle, or it allows the resource to be shared among multiple threads during a single cycle. We refer to these as *bandwidth-partitioned* and *bandwidth-shared*. These are new terms that are coined here to provide greater clarity when both mechanisms and policies are being discussed. These are mechanism terms. When there is no room for confusion, we will often refer to them simply as *partitioned* and *shared*. They have a close relationship to policy granularities, discussed below.

Generally speaking, if a single thread contains enough operations to completely consume a bandwidth resource (i.e., a pipeline stage) during a given clock cycle, then it is easier to implement a bandwidth-partitioned method where all the resource is given to the one thread. This is evident in Pentium 4 hyper-threading where most of the stages are bandwidth-partitioned. There are important exceptions to this, however, as will be discussed later in this chapter.

CAPACITY SHARING

A capacity resource contains some type of storage space that is assigned to be used by a thread. As such, a capacity resource may be consumed for some (possibly long) period of time. It is typically buffer or memory space (including cache memories). Total buffer capacity can be shared among threads, using two principal mechanisms. In one, the resource is *capacity-partitioned* so that a thread is given a defined physical portion of the resource and only that thread can use its defined portion. The other principal mechanism allows the resources to be *capacity-shared* so that buffer entries can contain any mix of threads. These will sometimes be called *partitioned* and *shared* for brevity. The term *pooled* is another way of saying shared. Thread identifiers (see Figure 17) are a mechanism that allows pooled resources.

Partitioning mechanisms can be either *static* or *dynamic*. Static partitioning is designed-in so that the capacity given to each thread cannot be changed. The queues in the Pentium 4, for example are statically partitioned. Dynamic partitioning allows the partitioning to be changed, possibly under software control, depending on the thread(s) that are running. It is a matter of policy to decide when to change partitioning and by what amounts.

Shared or pooled capacity resources may have additional mechanisms that enforce an upper bound on the amount of resource that a given thread may have. This mechanism could be used, for example, with a policy that attempts to assure a level of fairness or performance isolation.

As an example of both bandwidth and capacity mechanisms, consider again the Pentium 4. Figure 19 illustrates resource management in the Pentium 4 pipeline. Across the top are the implemented mechanisms for each of the bandwidth and capacity resources. Later, we will discuss the associated policies (along the bottom of the diagram).

© 2007 by J. E. Smith

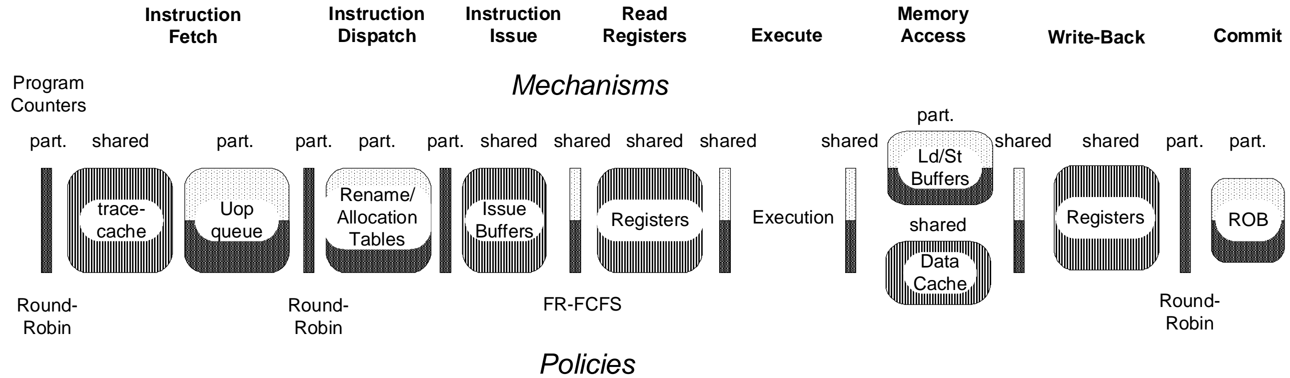


Figure 19. Pentium 4 hyper-threading mechanisms and policies.

PRE-EMPTION

Pre-emption is a mechanism whereby a thread is currently using a resource, but can be pre-empted so that another thread can use the resource. When this occurs, the initial thread, must re-acquire the resource and begin again. This could happen for example, if a buffer can be flushed of all entries belonging to a given thread. Or, a serial divider may be preempted. An alternative is to pre-empt by moving the entries to a holding buffer while another thread uses the main buffer. Such preemption might be a useful mechanism for policies that give threads priorities. A high priority thread may pre-empt a low priority thread, for example.

FEEDBACK MECHANISMS

All mechanisms don't have to be controlled by policies; others can monitor resource activity and provide feedback to a policy for making control decisions. Hence, feedback mechanisms are often closely tied to policies, as we will see later. Feedback allows a policy to consider prior behavior regarding resource usage in order to determine what to do in the future. A feedback mechanism may, for example, measure the issue rates for the different threads or measure buffer occupancy for the threads. This information, when provided for policy decisions, may be useful for optimizing overall performance or assuring some degree of fairness.

3.4.3 Policies

Now we consider policies that are implemented via the above described mechanisms. We first consider some higher level issues regarding policies in this section. Then, in subsequent sections we consider specific policies applied to both in-order and superscalar out-of-order pipelines.

POLICY COORDINATION

In theory, the resource allocation policies that are implemented in the various pipeline stages may be designed independently, but in practice, they are designed so that they work together well. Individual resources are subject to local policies, but they work within an overall, global policy. See Figure 20. This does not mean that there is necessarily physical hardware and connections between all the blocks of hardware that implement the local policies. In some cases the global policy may merely be an overall strategy to which the local policies contribute. There is often a natural correlation between the mechanisms. For example, the policy at the fetch stage of a pipeline generally influences all the downstream policies. With a thread partitioned mechanism and a one-wide in-order pipeline, for example, it is typically the first stage in the pipeline where the scheduling policy is actually implemented. The other, downstream stages simply process the instructions they are given. The first stage (typically the I-

© 2007 by J. E. Smith

fetch stage) implements a basic scheduling policy, and the downstream stages implement policies that complement the I-fetch policy.

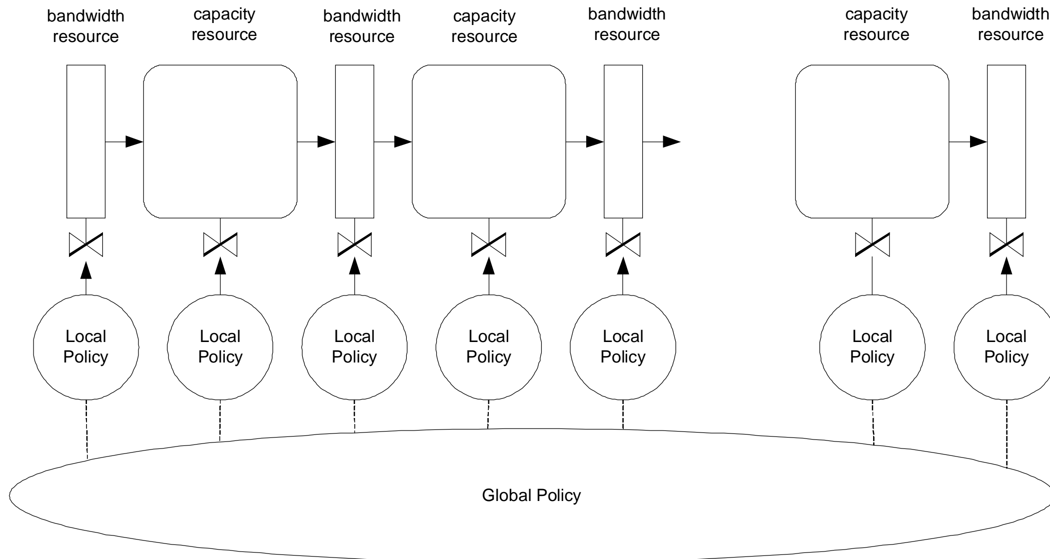


Figure 20. Local policies manage local resources in accordance with a global policy.

There may also be feedback from later stages that provide input to policies at frontend pipeline stage(s) (see Figure 21).

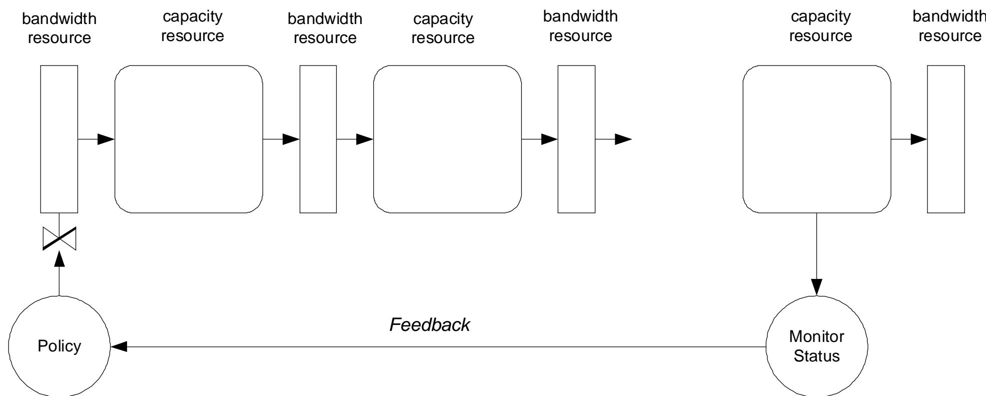


Figure 21. A policy may incorporate a feedback mechanism that monitors the status at a later pipeline stage.

SCHEDULING GRANULARITY

When refer to *scheduling policies*, we mainly mean allocation of bandwidth resources, for example certain pipeline stages (or perhaps the entire processor, at a macro-level). With regard to scheduling policies, two main components are *when* to make scheduling decisions, what we call the *scheduling granularity*, and *which* threads should be allocated a given resource when a scheduling decision is to made. We refer to this as *thread selection*. Scheduling granularity in this subsection and thread selection in the next.

Consider an individual pipeline stage in a superscalar processor. A policy schedules the stage at *coarse granularity* if one thread gets to use the stage for multiple cycles before the pipeline stage is given to another thread (See Figure 22a). This policy is ordinarily used in conjunction with a bandwidth-partitioned mechanism for the pipeline stage. In practice, if coarse granularity is used in one stage, it is

used in all the stages of the pipeline. Primarily, scheduling is controlled at the instruction fetch stage. The scheduler decides which thread gets to use the fetch unit and then this thread uses the fetch stage (and all the other stages) until the scheduler switches to another thread. Then, downstream pipeline stages see the same blocks of instructions belonging to individual threads.

A second general policy for scheduling threads is *fine granularity* (See Figure 22b). Here there is potentially a thread switch at every clock cycle, but a given pipeline stage is used by only one thread at a time. That is, the mechanism is bandwidth-partitioned. The Denelcor HEP and CDC 6600 PPs used fine-grain scheduling policies throughout. As we have seen, some (but not all) stages of the Intel Pentium 4 also use a fine grain policy.

We see that fine and coarse grain multi-threading are different policies that use the same mechanism (e.g., a bandwidth partitioned mechanism). Moreover, it sometimes the case that if a coarse-grain scheduling policy is implemented, the underlying mechanisms may be specialized so that they can not support a fine-grain policy. For example, a processor with a coarse grain mechanism may contain non-partitioned buffers that cannot be shared among threads (e.g., TIds may not be implemented). In this case, it is assumed that at any given time, all the buffer entries belong to one thread.

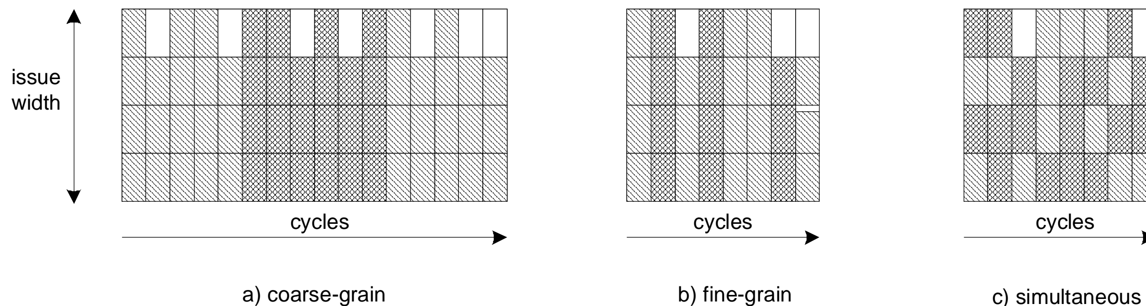


Figure 22. Multi-threaded scheduling policies.

The third policy for scheduling threads – one that is specific to superscalar processors – is *simultaneous multi-threading* (SMT). SMT is applied on top of bandwidth-shared mechanism and is illustrated in Figure 22c. With SMT, the same pipeline stage can be given to multiple threads during the same cycle. For example, in the Pentium 4, the issue stage can issue instructions from more than one thread at the same time.

Also, as noted above, the scheduling policies may be mixed within a given design – the Pentium 4 (see Figure 19) uses both fine-grain and simultaneous scheduling policies at different pipeline stages. However, with coarse-grain policies, all the stages tend to use the same policy.

THREAD SELECTION

The second important aspect of scheduling policies is which threads to select when a scheduling point is reached. There are a number of possibilities. Some of the basic ones are listed here; other, more elaborate policies are discussed later.

Round-Robin – a round-robin policy rotates through the threads when it selects which should be assigned a bandwidth resource. If a given thread does not require the resource during a cycle it would be selected, then the round-robin goes to the next thread that does have an active request. The Barrel and Slot mechanism of the 6600 PPs is a good example of Round-Robin scheduling.

© 2007 by J. E. Smith

LRU – a least recently used policy selects the thread that has least recently used a given resource. If all the threads are always requesting a resource, then LRU degenerates to round-robin. However, if a thread has been has not made requests for a number of cycles (for example, if the thread has suffered an instruction cache miss), then it has highest priority immediately when it re-enters the selection process.

FCFS – a first-come, first-served policy would typically be applied when threads are selected from a buffer. With this policy, the first one to enter the buffer is the first one selected; it is equivalent to *FIFO*.

FR-FCFS – First ready, first come first served is a special case of FCFS that it accounts for the fact that all the operations in a buffer may not be ready for service. Hence, it may not make sense to select the oldest item, rather the policy should select the oldest of the ones that are ready to use the resource. This is FR-FCFS. This policy often used in an instruction issue buffer, for example, where instructions must be free of register dependences before they are ready to issue; of the ones that are dependence free, the oldest ones are selected. This is the policy used in Pentium 4 hyper-threading.

Priority – priority policies select threads according to a specified priority, selecting high priority threads ahead of low priority threads. This policy would typically be based on a mechanism whereby priorities can be set by software. This might be done via a control register, for example. If it is possible for two threads to have the same priority, then a secondary policy, for example round-robin may be used for selecting among threads of the same priority.

WORK CONSERVATION

Generally speaking, processor resources are able to perform *work*, where work is defined as either processing (for a bandwidth resource) or storage (as in the case of a capacity resource). A policy for managing a shared a resource is *work conserving* if none of a resource goes unused (or is wasted) as long as some thread can use that resource. There is a connection between work conservation and mechanisms. If a resource has a static partition, for example, then the allocation policy is wired into the mechanism, and, in this case, it is usually not work-conserving. For example, one might implement a mechanism that partitions resources by assigning a fixed fraction of the resource to a given thread, as is done in the Pentium 4 for some of the buffer resources. If one thread does not use all of its resource, no other thread can use that resource. If some other processor cannot use it, then work gets wasted (it is not work conserving). Generally speaking shared resources may permit work conservation (depending on policy) and partitioned resources do not.

As another example, in some pipeline stages a fine-grain scheduling mechanism may not be work conserving while a simultaneous scheduling mechanism is work conserving. Consider the issue stage of the Pentium 4 processor. If the issue stage could only issue instructions from one of the threads in a given clock cycle, then some issue slots may go unused. The Pentium 4 designers chose simultaneous scheduling in the issue stage to be work conserving.

CAPACITY POLICIES

The capacity resources are buffers and caches. Shared cache resources will be discussed in greater detail in Chapter 5 because multi-core chips often share caches regardless of whether the cores are multi-threaded (and in this chapter we are focusing mainly on multi-threaded cores). Buffer resources may be partitioned or shared. If they are partitioned, then there is little room left for policy; each thread simply uses its assigned portion of the buffer.

If the buffers are shared, then some policies do come into effect. Because of fairness considerations, for example, a policy may enforce some maximum threshold of entries, even though the buffer is shared. This will avoid one thread completely occupying the buffer (and then stalling and preventing other threads from making forward progress). Partitioned buffers avoid this problem, of course, and this is the approach taken for many of the buffers in the Pentium 4, as we have seen.

The amount of a capacity resource given to a thread may affect its bandwidth demands for the next, downstream resource, and this effect may be relatively complex. The primary example is the issue buffer and its effect on issue rate. If a thread is given more of the issue window slots, then it will likely be able to achieve a higher issue rate, but, as we have observed this is not a linear function; it is closer to quadratic. Cache memories provide another example. The more cache entries that a thread is given, the lower its miss rate will be (See Figure 9 of Chapter 1); this is discussed more in Chapter 5.

3.4.4 Scheduling Policies for In-Order Processors

We now consider specific scheduling policies for processors with in-order pipelines. In-order pipelines usually use either fine-grain or coarse grain scheduling, but not simultaneous. This is obvious if the pipeline is only one-wide. But it has also been the case for superscalar in-order processors. Generally, in-order processors are implemented because they are simpler than out-of-order processors, and, avoiding simultaneous mechanisms is in line with the overall philosophy of simplicity.

A key issue in resource scheduling is reducing the amount of wasted resources. Consider the performance profiles for two threads being processed in in-order pipelines as shown at the top of Figure 23. These figures show the cycles where instructions are executed and where they are stalled either because of instruction dependences (labeled "pipeline stalls") or data cache misses. The stall and cache miss cycles represent wasted computational resources. If the two threads are scheduled at fine granularity with a round-robin policy (the third sequence in Figure 23), then very few cycles are wasted. With round-robin scheduling, instructions from one thread tend to fill-in the unused, stall cycles of the other. In a few cases, however, where both threads stall for cache misses, some cycles are wasted. If cache misses are longer than shown here, for example, then the number of wasted cycles would grow. In this case, one could potentially introduce additional threads to fill-in the gaps. In general, one should have enough threads that wasted cycles are relatively rare.

A coarse-grain scheduling policy is shown in the fourth sequence of Figure 23. We observe that for the single threads, large blocks of wasted cycles occur when there is a cache miss. Consequently, cache miss events are selected as points where a thread switch should occur. This policy is referred to as *switch-on-event*. A switch-on-event policy may be simpler to implement than fine-grain scheduling, and as long as events such as cache misses are the dominant cause of wasted cycles, it will be fairly efficient in filling the gaps. In this example, the cache miss latency is relatively small (about 6 cycles). Depending on the cache hierarchy, and which caches suffer misses, the blocks of wasted cycle may be much larger, which makes switch-on-event even more effective, relatively speaking.

© 2007 by J. E. Smith

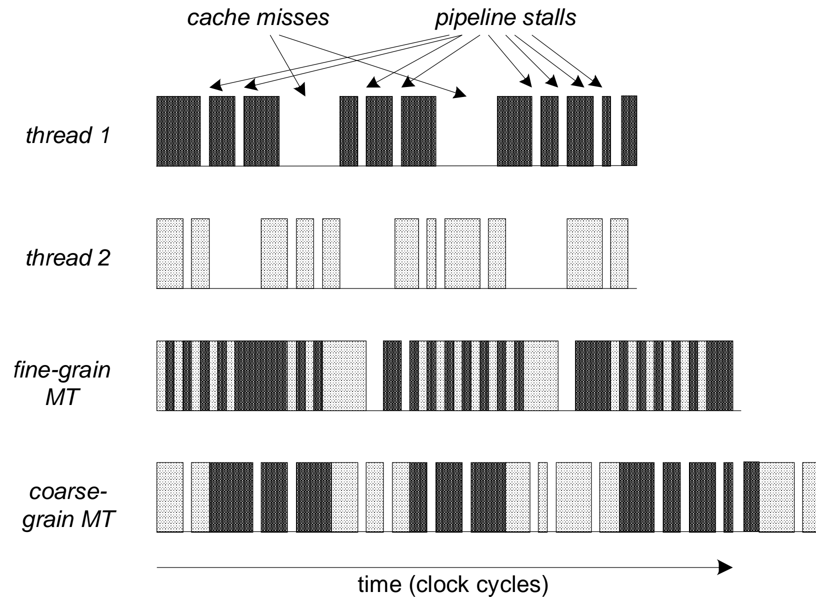


Figure 23. Scheduling of two threads with fine-grain round-robin scheduling and coarse-grain switch-on-event scheduling.

Usually with switch-on-event there is also a timeout mechanism so that a given thread has a maximum time interval (or number of instructions) before there is a thread switch. This is to assure degree of fairness.

Switch-on-event is an example of a policy that uses a feedback mechanism. For example, a data cache miss is an event that occurs several pipeline stages downstream from the fetch stage. So, there is feedback from the cache miss stage to hardware implementing the thread selection policy at the fetch stage. When there is an I-cache miss, the feedback path is much shorter. Depending on the number of stages that the feedback spans, there may be delay in starting up the next thread after a switch. For example, in a relatively short in-order pipeline it may take two or three cycles before instructions from the new thread begin executing. Hence the switch-on-event profile in Figure 23 is optimistic. In a real implementation there would be a gap of unused cycles, e.g., three, at the time of a thread switch. This is illustrated below when we consider the IBM RS64 IV as a case study (in Section 3.5.1)

An in-order pipeline with a fine-grain scheduling policy is often used in applications where aggregate throughput is the objective, not the performance of individual threads. In this situation, the already simple pipeline design can be simplified even more by removing some or all of the forwarding logic (see Figure 24). When this is done, there are more stalls when a thread runs by itself, so the thread goes slower than when there are forward paths (see the top two profiles of Figure 24). However, fine-grain multi-threading can still be effective in filling in the gaps (see the bottom profile of Figure 24 where a fine-grain round-robin policy is used. This is essentially the approach taken in the Denelcor HEP system discussed above.

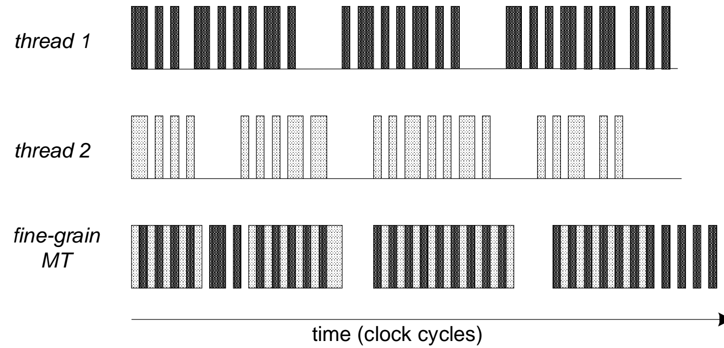


Figure 24. Fine-grain multithreading of pipelines without forwarding hardware. There are more gaps due to stalls in the individual threads. However, fine-grain multi-threading is able to fill in most of the gaps.

3.4.5 Out-of-Order Scheduling Policies

Now we consider scheduling policies in multi-threaded out-of-order, superscalar pipelines.

SCHEDULING GRANULARITY

Superscalar pipelines tend to be scheduled with some mix of fine and simultaneous granularity; coarse granularity is not used. To understand why, consider the performance profile of a superscalar processor (Figure 25). In superscalar processors there is relatively little performance loss due to instructions waiting for ordinary register dependences involving short-latency instructions; rather there are performance gaps due to branch mispredictions and cache misses. Conceptually, a coarse-grain switch-on-event policy would switch threads when one of these miss events occurs.

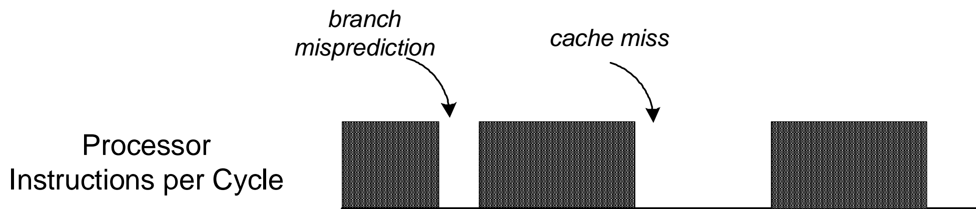


Figure 25. Superscalar processors instruction execution is interspersed with miss events (branch mispredictions and cache/TLB misses).

Now consider the performance effects that a switch-on-event policy would have. This should be done while keeping in mind that superscalar pipelines are usually much longer than in-order pipelines; there may be ten stages or more prior to instruction execution. Then, for branch mispredictions, switch-on-event would provide no advantage. When a misprediction is detected, the pipeline has to be flushed and refilled beginning with instruction fetch. It makes no difference if the old thread is restarted or there is a switch and a new thread is started; there is still a gap in performance equal to the pipeline depth.

For a short latency, L1 data cache miss, the front-end pipeline length is about the same as the time it takes to handle the L1 miss. Again, there would be little advantage to starting a new thread. First, there are probably enough instructions from the old thread already in the issue buffer to overlap most of the miss latency, and in any case, the data would be back from the L2 by the time instructions from the new thread arrive at the issue buffer.

So, that leaves L1 I-cache misses and longer miss events such as L2 cache misses and TLB misses where switch-on-event would provide some advantage. And, even for these, the long pipeline front-end would translate to a long effective switch time, which would significantly diminish any perform-

© 2007 by J. E. Smith

ance advantage. A recent study Gabor et al. [10] regarding switch-on-event in superscalar processors only switched on the event of an L2 cache miss, where the miss latency was 300 cycles. This means that all the performance gaps due to branch mispredictions and L1 cache misses are not filled in by multi-threading.

Coarse grain scheduling has been used with in-order superscalar pipelines, which tend to have much shorter pipeline front-ends. For example, this is the approach taken in the IBM RS64 IV, which has a three cycle switch penalty and is described below in Section 3.5.1. The Intel Montecito [38] is another superscalar in-order implementation that uses a switch-on-event policy. It only switches on long latency miss events such as an L3 cache miss to memory, as well as some other events not related to misses.

An approach which might make switch-on-event more feasible is to predict the events in advance. This can be done, for example, by keeping track of branches that are more likely to be mispredicted [28]. If these are used as thread switch points, then even if the prediction is incorrect, performance wouldn't necessarily be lost.

For out-of-order superscalar processors, fine-grain scheduling policies in the pipeline front-end would inevitably lead to mixed instructions from multiple threads in the issue buffers, yielding some useful work to do be done by one (or more) other threads when a thread suffers a branch misprediction or an L1 cache miss. The performance while the miss event is being handled may not be at maximum throughput, but other threads would make some progress.

So, if we consider fine-grain and simultaneous scheduling policies at the various superscalar pipeline stages, many of the stages are best scheduled with fine-grain policies (bandwidth-partitioned) because one thread has enough work to occupy the full bandwidth of the stage. This is often the case for the front-end stages (and is the approach used in the Pentium 4, for example). On the other hand, it is most efficient to use shared resources at the issue buffer; the issue buffer and associated logic is an expensive, complicated piece of hardware, and because of quadratic growth property and other practical reasons, it is more efficient to share the issue buffer resource. Then, once the issue buffer is shared, it is probably easier to issue instructions with a simultaneous granularity policy than a fine-grain policy. In fact, it would probably take extra logic to perform fine-grain scheduling rather than simultaneous scheduling; this logic would be there to make sure that all the instructions are from the same thread because the issue decisions tend to be made locally. Most multi-threaded out-of-order superscalar processors issue instructions simultaneously without regard to the threads to which they belong.

SINGLE THREAD POLICIES

In a number of multithreaded processors, one objective is to achieve high performance when only a single thread is active; or, stated another way to give all the resources (or at least all the resources a thread could possibly use) to a single thread when there are no other threads. The reason is that there are either situations (or operating modes) where there is only one thread with work to do. A multi-threaded processor may have a software-settable mode bit that either enables or disables multi-threading. When disabled, the processor acts as a high performance single-threaded processor. This single-thread performance objective affects the implementation of mechanisms and policies, as well as some processor resources.

One important effect is on sizing of resources that grow quadratically with instruction issue rate (Figure 13). Even though all of these resources do not grow exactly quadratically for all programs, they do grow super-linearly for very many programs. For sake of discussion, we assume that they grow

quadratically, however. Also, for sake of discussion, we will focus on the issue buffer, but a similar argument holds for the other quadratic-growth resources.

Say there are *n* threads and a superscalar issue width of *d*. Then, it can be shown that the minimum issue buffer size that will achieve issue rate *d* is one where each of threads issues at rate *d/n*. Assuming the quadratic relationship, the total issue buffer size is $n*(d/n)^2$; each of the threads needs a buffer of size $(d/n)^2$ to sustain its issue rate and there are *n* such threads. On the other hand, if one thread is to sustain an issue rate of *d*, then the total buffer size is d^2 . Figure 26 illustrates these relationships for different numbers of threads and a total issue bandwidth of eight. If there is only one thread, then the issue buffer is size 64. At the other extreme, if there eight threads then, at least in theory, the issue buffer only has to contain eight instructions (keep in mind that the quadratic growth relationship is for instructions with unit latency.) For a more realistic two threads, the issue buffer required for sustaining an issue rate of 8 is half the size required for one thread.

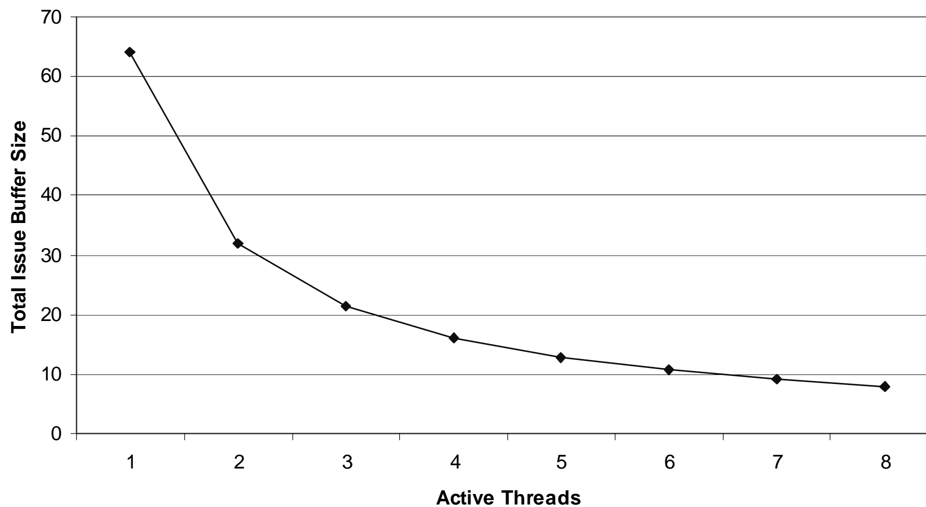


Figure 26. Relationship between the number of active threads and the aggregate issue buffer size.

This illustrates a significant cost issue when a performance objective is to support a single thread at maximum performance -- the issue buffer (and all other quadratic growth structures) are larger than if the maximum single thread performance objective is removed. This effect is more pronounced for wider issue widths. For wide pipeline widths, the single thread performance objective can also make instruction fetching difficult. This is discussed in the next subsection.

FETCH UNIT MECHANISMS AND POLICIES

As noted earlier, a policy at one pipeline stage often affects (or even forces) the policies at downstream stages. Consequently, multi-threaded superscalar processors tend to implement policies at certain key pipeline stages, and these policies then directly imply policies at other stages. In many processor designs, the key pipeline stages are instruction fetch, instruction issue, and retirement. The fetch policy heavily influences policies in stages prior to the issue buffer. The issue policy influences the execution, memory, and write-back stages, and retirement, being at the end of pipeline doesn't influence other stages directly, but it often differs from the issue policy. In this subsection, we consider instruction fetch mechanisms and policies, and then in the following two subsections we consider instruction issue and retirement.

As noted above, because the fetch unit injects instructions into the pipeline, the fetch unit policy typically sets the tenor for the policies downstream in the pipeline, prior to the issue buffer. For conventional instruction cache-based fetch units, the wider the fetch width, the less likelihood that the fetch unit will be able to fetch a block of contiguous instructions from the same thread. The average instruction fetch width that can be sustained for a single thread is a function of the cache line size, but it is also a function of where conditional branches and branch targets fall within a line, as well as the complexity of a fetch unit that may integrate branch prediction so that instructions can be fetched beyond non-taken branches.

For narrow pipeline widths (say four or less), it is fairly easy to fetch blocks of instructions from a single thread that fill the pipeline width. Although this is often done with a fetch buffer to smooth out the instruction flow by averaging fetch rates over multiple cycles. Most of the implemented multi-threading designs have pipeline widths of four or less, so the fetch stage must only support a mechanism that fetches from one thread in a given cycle. Hence, it is natural to use bandwidth-partitioning and fine-grain policies for instruction fetching, and this is typically done.

Fetching larger numbers of instructions per cycle, say up to eight, on a sustained basis may require fetching past predicted taken branches [39]. This would require a relatively complex fetch unit, combined with a multi-ported instruction cache. Another option is a trace cache, which contains dynamic instruction blocks with embedded taken branches [36]. An early study of simultaneous multi-threaded processors looked at the instruction fetch issue. Using a fairly simple fetch unit that could supply at most eight instructions from a single cache line, the authors concluded that it was necessary to fetch from two threads to satisfy an eight wide SMT processor

More sophisticated instruction caches that can fetch contiguous instructions from two adjoining lines and/or that can fetch past predicted not-taken branches can be constructed. However, these are not capable of fetching past predicted taken branches, and this will be a limiter in a significant number of programs, so the basic point made in the SMT study [36] remains valid. Building an instruction fetch unit capable of going beyond taken branches entails either something like a trace cache, or a multi-ported instruction cache with an integrated branch prediction approach.

This has led to fetch unit designs for wider pipeline widths (e.g., eight) that are capable of fetching from two threads simultaneously [5]. This also requires multiple (two) instruction cache ports (although this is probably simpler than multiple fetches from the same stream with the need for branch prediction to tie the two fetches together). However, it does not achieve the single thread performance objective stated above.

A conclusion is that for a simple fetch unit, if the pipeline width is four or less, then the fetch stage can be partitioned, fetching from one thread at a time; otherwise it has to be shared (with the additional instruction cache and fetch unit complexity). Most multi-threaded superscalar processors have been of width four or less; the IBM Power5 is capable of width five, but this depends on groups of instructions satisfying a number of constraints, so a width of five is not sustained in practice.

INSTRUCTION ISSUE POLICIES

With respect to instruction issue policies, the main topic to be dealt with is the shared instruction issue buffer. A shared buffer is good because it allows more efficient usage of an expensive resource. On the other hand, imbalances in issue buffer occupancy (the number of issue buffer slots that a given thread holds) can lead to inefficiency in issue. This is a corollary of the tradeoff in issue width and buffer size discussed above; the best throughput is achieved if instruction buffer slots are used in a balanced manner. This leads to policies that attempt to balance the number of issue buffer slots con-

sumed by the active threads. Not only does this lead to better aggregate performance, but it also tends to a higher degree of yield fairness.

Implementing such a policy implies a feedback mechanism. In the early study on SMT processors [14], the authors considered a number of policies targeted at increasing aggregate issue rate. The one that appeared to work best balances the number of un-issued instructions following the fetch stage (these are instructions being decoded, renamed, or in the issue buffer). There is a mechanism for counting the number of outstanding, un-issued instructions for each thread. This feeds into a policy that gives fetch priority to threads with lower counts. The authors refer to this policy as *ICOUNT*. Figure 27 shows results for ICOUNT when compared with round-robin fetching. We see that performance is improved significantly with ICOUNT. ICOUNT tends to balance the resource usage among outstanding, un-issued instructions. This avoids the situation where one (or a small number) of threads occupy most of the issue buffer resources, thereby starving the other threads and reducing overall throughput.

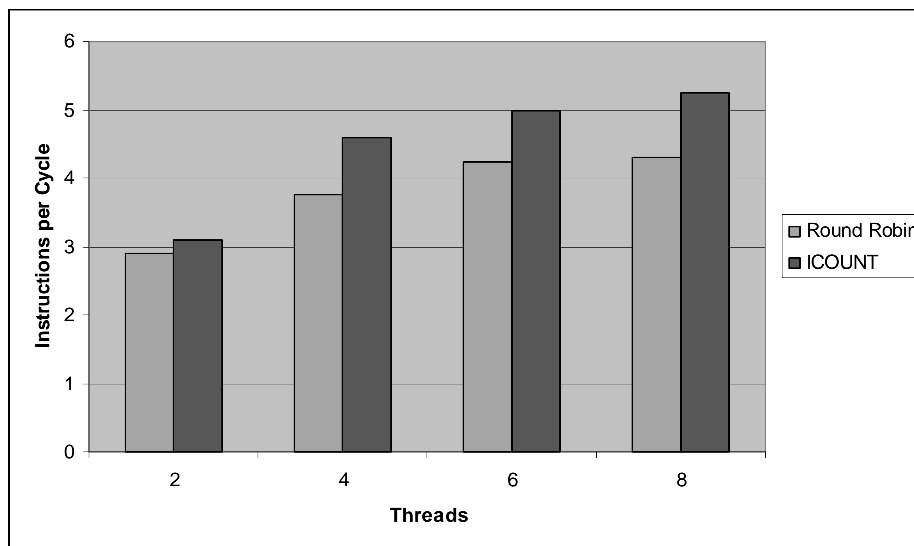


Figure 27. Performance comparison of Round-Robin and ICOUNT fetch policies in an 8-way SMT prrocessor (from [reference]).

RETIREMENT POLICIES

The retirement policy is not performance critical; there are no downstream pipeline stages that depend on it. So, for design simplicity, retirement tends to be fine-grain, not simultaneous. Note that the ROB is normally managed in a strict FIFO fashion. This implies that a straightforward design approach is to partition the ROB, and manage retirement from the partitioned ROBs in a round-robin fashion. If the ROB is not partitioned, then its resources will have to be managed in a more complex fashion; see the IBM Power5 discussion below.

FAIRNESS POLICIES

Thus far, we have considered policies that are implicitly or intuitively fair. There are also policies that attempt to meet a quantifiable fairness objective. As described in Section 3.4.1, measuring harmonic mean speedups provides an intuitive level of fairness with an aggregate performance measure. For example, a very unfair method, that for example starves one of the threads, would have a low harmonic mean speedup. Because of the way it is defined, this measure is not applied directly when implementing a policy. Rather, it is only used during the evaluation process. That is, a designer (or researcher)

© 2007 by J. E. Smith

may propose a scheme for sharing resources, then, by simulating it and measuring performance with the harmonic mean, fairness is implicitly included in the evaluation.

A more explicit fairness measure is proportionate loss, where the objective is for all the concurrently running threads to have the same speedup, where speedup is measured with respect to the thread running alone with all the hardware resources. There have been proposals for building policies and mechanisms targeted at this fairness measure. The major difficulty is in determining the performance a thread would have if running alone. One approach is to use off-line profiling of a task to arrive at its isolated performance. Then, when it executes concurrently, this information is conveyed to the sharing policies. This can be done indirectly by estimating the required resources for achieving proportionate speedup [23]. Another approach provides a feedback mechanism that can monitor the concurrently executing threads to estimate what their performance would be, if running alone. Then this feedback information provides input into a fetch unit thread selection policy [10].

Another definition of fairness is based on fair resource allocation rather than fair performance measures. That is, each thread gets a fair share of resources; or in general, some allocated fraction of resources. Fairness is the case where each of *n* threads gets *1/n* of the resources. This resource based notion of fairness was studied for certain capacity resources, e.g., the issue buffer. In that work [21] it was shown that partitioning the issue buffer does not degrade performance significantly (versus dynamic sharing), and it makes the instruction fetch selection policy less significant. For example, a round-robin selection policy works nearly as well as the ICOUNT policy.

EXPLICIT PRIORITY POLICIES

The previous section considers policies that attempt to achieve fairness. However, there are other policies where, in effect, the goal is unfairness. That is, policies where certain threads are given higher priorities than others, and therefore receive more of the resources. This might be done, for example in the case of a soft real-time thread, or an interactive thread where response time is important. In general priorities are more often applied to bandwidth resources. For example, a higher priority thread may get a higher fraction of the fetch cycles (and, therefore, the other pipeline cycles).

If capacity resources are assigned priorities, then one might give a larger fraction to higher priority tasks, although one still might want to guarantee a minimum level of resource for low priority tasks so that they can always make forward progress. A priority scheme used in the Power5 is described in Section 3.5.3.

3.4.6 Macro-Level Scheduling Policies

Although most hardware resources in a multithreaded processor are shared via hardware mechanisms and policies, the OS does have an indirect ability to manage resource sharing by deciding which threads to run concurrently. Because different threads have different resource requirements, some combinations of threads may compete much more for resources than other combinations of threads.

This observation leads to what has been termed *symbiotic jobscheduling* [30]. With symbiotic job scheduling, the OS runs an over-head free sampling phase where it collects information regarding programs. Then, using this information, it decides which should be scheduled to run concurrently in order to optimize overall throughput.

3.5 Multi-Threaded Cores: Case Studies

We consider three important case studies which illustrate the spectrum of multi-threaded processors. All three are real industrial designs (although one of them never shipped as a product). One of them uses switch-on-event, one uses fine-grain, one uses simultaneous scheduling.

© 2007 by J. E. Smith

3.5.1 IBM RS64 IV

The IBM RS64 IV is a PowerPC implementation used in IBM pSeries server systems. It was not used in any client (desktop) systems, and this allowed the designers to focus on server applications. In server applications there is usually an abundance of thread-level parallelism, either through independent threads or server applications that have already been optimized for parallel processing. To the OS, the multi-threaded RS64 IV looks like a logical multiprocessor.

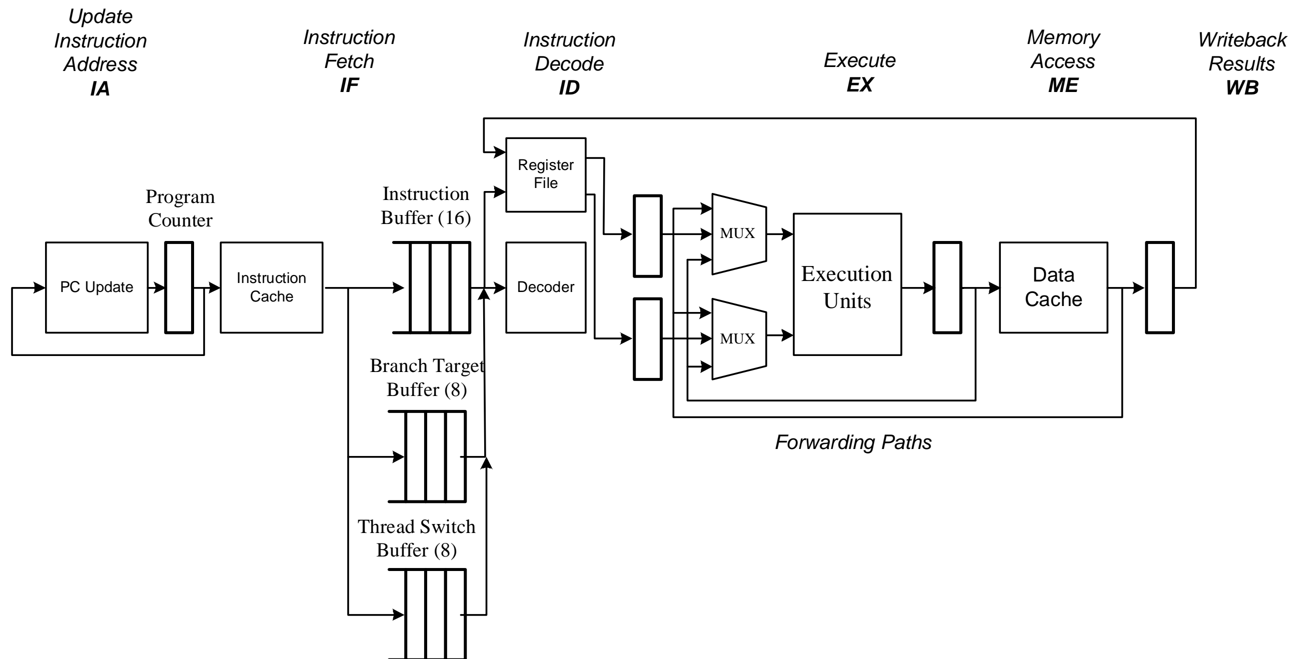


Figure 28. The IBM RS64 IV pipeline has conventional in-order pipeline stages. It is a 4-way superscalar processor and has instruction buffers to reduce branch misprediction and thread switch delays.

The processor supports two threads, with duplicate architected state for the threads; that is, there are separate per thread register files and program counters as well as other PowerPC architected registers such as the condition and link registers. The processor is 4-way superscalar with a 5-stage pipeline, and it issues instructions in-order (Figure 28). Because it is an in-order pipeline there is relatively little buffering in the pipeline (as compared with Pentium 4 hyper-threading, for example).

The RS64 IV processor uses a switch-on-event fetch policy combined with priorities. With switch-on-event, a foreground thread is executing while a background thread is waits. The foreground thread executes with all the resources until there is a long latency event such as a cache miss. Then, there is a thread switch to the background thread (which then becomes the foreground thread).

Priorities can be set by the OS. For example, a thread in an idle loop may be given a low priority. This is done via a thread State Control Register (TSC). This register holds the conditions that will trigger a thread switch, and priorities are implemented by adjusting the events that will cause a thread switch. These include L1 cache misses for both instructions and data, L2 misses for both instructions and data, TLB misses, and when a timeout counter value is reached. There is also a hierarchy among the switch events. Because an L2 cache miss is likely to take longer than an L1 miss, if one thread encounters an L1 miss while the other thread is waiting for an L2 miss, then the thread with the L1 miss does not switch. The reasoning is that its L1 miss will be handled before the L2 miss in the other thread is

complete. As an optimization, an I-TLB miss triggers a prefetch into the instruction cache to avoid the I-cache miss that is likely to occur after the I-TLB miss.

The Thread Switch Timeout (TST) register specifies a value when a thread switch is forced, to assure a level of fairness in the event that the foreground thread does not encounter any of the thread switch events for a long period of time. Hardware also maintains a forward progress count that is incremented whenever the thread encounters a thread switch (is switched out). This count is reset to zero whenever the thread completes an instruction. If this count should exceed a value held in the TSC, then thread switching is disabled until this thread can complete at least one instruction. This guarantees that each thread makes some forward progress.

To reduce thread switch delay, the fetch unit maintains a thread switch buffer (shown in Figure 28) which contains pre-fetched instructions from the background thread, ready to go when a thread switch occurs. The instruction fetch unit is capable of fetching up to eight instructions per cycle, so it can run ahead of the four-wide pipeline, and then use extra fetch cycles to fill the thread switch buffer. The fetch unit also prefetches alternate branch paths when branches are predicted; these are used to reduce the branch misprediction penalty.

Figure 29 illustrates a thread switch on L1 D cache miss. This example shows only a single instruction per cycle for clarity. The first instruction, a load from thread 1 is fetched and is held in the fetch buffer for some number of cycles (we are assuming that the fetch unit has already run ahead of the rest of the pipeline. Then, the second instruction from the same thread is fetched, and eventually starts down the pipeline. Then, as illustrated in the figure, instructions from the second, background thread are fetched and placed in the thread switch buffer. The load instruction misses in the data cache; this is not detected until the writeback (WB) pipeline stage. This miss causes the pipeline to be flushed of instructions following the load, and a thread switch takes place. Now, instructions can be read from the thread switch buffer, so the first instruction from the second thread can be decoded the cycle after thread 1's cache miss is detected. The net result is a three cycle thread switch penalty.

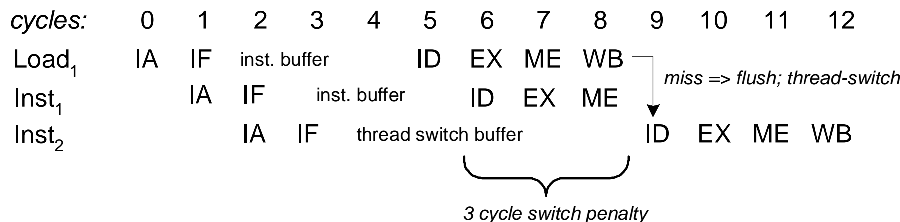


Figure 29. Thread switch timing on a data cache miss. The processor is 4-way superscalar, but to simplify the figure this is not shown.

Figure 30 illustrates causes of thread switches for a set of benchmark programs. The causes are more-or-less as one would expect. Most of them are due to L1 cache misses (both instructions and data). Timeouts account for 17%, indicating that the threads to experience long runs of instructions with no miss events.

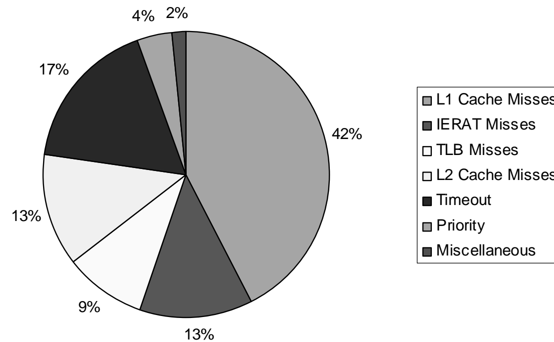


Figure 30. Causes for thread switches. The IERAT serves as an instruction TLB.

With respect to cost, the designers estimate the additional chip area for multithreading to be about 5%. They say that the affect on clock cycle is negligible. This is probably in part due to the switch-on-event policy, where there is no need for control decisions at most of the pipeline stages. That is, most of the pipeline is designed exactly as it would be for a single threaded design. The designers estimate that the design time was lengthened by about two months in order to implement multi-threading.

3.5.2 SUN Niagara

The SUN Niagara processor has an in-order pipeline with fine-grain scheduling [12]. The Niagara processor is used in server systems targeted at throughput multiprocessing. In the first generation, a single chip contains eight processor cores, each of which can support four threads. Besides the cores, themselves, each core has its own L1 caches (16K I cache, 8K D cache). There is a single, shared floating point unit, reflecting an emphasis on commercial applications, which have relatively light floating point demands. In addition to the private caches, the chip also contains a shared 3 Mbyte L2 cache and memory controllers. Here we will focus on the Niagara processor. Chapter 6 on server MPs, includes discussion of the complete Niagara server system.

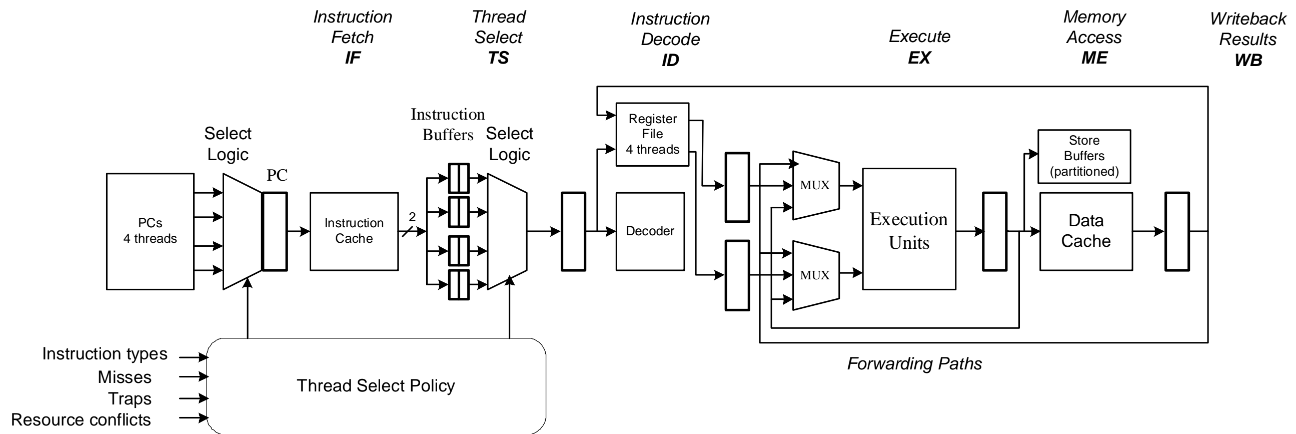


Figure 31. Block diagram of SUN Niagara multi-threaded processor pipeline.

A single Niagara processor core contains an in-order pipeline that can support four threads. A block diagram is in Figure 31. The pipeline contains forwarding logic so that ALU operations can be forwarded to immediately following instructions. Load data can be forwarded to instructions with a two cycle latency. The load/store unit contains separate store buffers per thread. Loads are checked

against addresses of pending stores, and data is forwarded from the store buffers when there is a match.

At the beginning of the pipeline, the PC for one of the threads is selected, and either one or two instructions are fetched from that thread and buffered (depending on whether the fetch address is an odd or even word address). Among other things, fetching ahead assures extra I-cache time slots for background cache line-fills, when they are needed.

Thread selection takes place prior to the fetch and decode stages, and a common, coordinated policy manages this selection. If a thread is selected for decode, then the same thread is selected for instruction fetch. In general, thread selection is performed on an LRU basis – the least recently selected active thread has priority.

At the time instructions are placed in the instruction cache, they are pre-decoded to identify long latency instructions; these instructions include multiplies, divides, conditional branches, and loads. The pre-decode information is fed to the selection logic, and a thread with a long latency instruction is temporarily deselected. Instructions from a thread that issues a multiply or divide are deselected until the multiply or divide completes (and its result data is available). Conditional branches are deselected until the branch outcome is known; hence, there is no need for branch prediction.

The handling of load instructions is a little more complex. Instructions in threads following load instructions are deselected at least until the time data would be available for bypass if there is a cache hit. If there are instructions from other threads ready to be selected, then instructions following the load wait an additional cycle until it is known whether there is a cache hit. Otherwise, an instruction from the load's thread issues speculatively, assuming a cache hit. If there is a cache miss, then instructions from the thread containing the load must be flushed (pre-empted), and the thread is deselected until the load data is available.

Figure 32 illustrates thread scheduling in a thread that contains a load instruction. This example assumes that only threads 0 and 1 are active. At cycle 0, a load instruction from thread 0 is selected and starts down the pipeline. The next cycle, using LRU scheduling, an add instruction from thread 1 is selected. At cycle 2, thread 0 is currently processing a long latency instruction (the load) and the load's result, even if it hits in the cache, will not be available for the next instruction in thread 0. Consequently, thread 0 is de-selected, and the next instruction from thread 1 is selected (a load instruction). Now, at cycle 3, thread 1 cannot be selected because of its load instruction. Then, thread 0 is the only choice. If the thread 0 load happens to hit, then the data will be available for forwarding to the add from thread 0, so the add from thread 0 is selected. However, this is still speculative, because at that point in time, it is not known whether the load will hit. As it turns out, the load does hit so its result is forwarded to the add. If, on the other hand, the load had missed in the cache, then the add from thread 0 would have been flushed from the pipeline, and thread 0 would be de-selected until the load miss is satisfied. If at cycle 3 there had been a third active thread, then thread 0 would have been deferred for a cycle until it was known that the load would hit in the cache (a speculative thread has lowest priority).

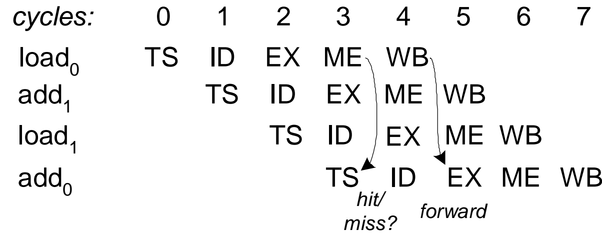


Figure 32. Example of Niagara thread scheduling. The add instruction from thread 0 is issued speculatively (assuming thread 0's load hits in the data cache).

3.5.3 IBM Power5

The IBM Power5 is a superscalar processor that employs simultaneous multi-threading. It is targeted at server systems. The Power5 is a follow-on design to the Power4, which was single-threaded. Each Power5 chip contains two multi-threaded cores. As with most other multi-threaded cores, a goal in the Power5 core is to support fast single threads as well as dual threads (single thread and SMT modes). In addition to the cores, the Power5 chip contains a 1.875 MByte shared L2 cache, and a tag directory for an off-chip L3 data cache, as well as a memory controller. In this section, we will focus on the cores; server systems will be discussed in Chapter 6.

A resource diagram is shown in Figure 33. This diagram shows the major resources (both bandwidth and capacity) and the policies that are used for managing them. The branch predictor is a pooled resource – its entries are shared by the two threads. This is a fairly complex predictor (it contains two types of predictors combined with a meta-predictor that chooses between the two predictions), and sharing it between the two threads is more efficient than would be the case with two separate predictors. On the other hand, return address predictors (not shown in the figure) are partitioned; these structures are small and their contents are entirely thread-specific.

Instructions are held in a shared instruction cache and are fetched from the threads in round-robin fashion. The fetched instructions streams are placed into two 24-entry fetch buffers. Instructions selected from the fetch buffers are then formed into *groups* for later tracking and retirement. To simplify the design, the Power5 tracks all the outstanding instructions in groups of no more than five instructions (rather than tracking them individually). There are certain constraints regarding the types of instructions that can be in a group, so it is not uncommon to have a group of fewer than five instructions. The group completion table (GCT) is a pooled resource; it is organized as a linked list to facilitate pooled multi-threading (rather than a FIFO structure as one might do with a single-threaded processor). Selection for group formation is done one thread at a time using a policy described below. Briefly, the policy is based on software-settable priorities as well as feedback from buffer monitoring mechanisms farther down the pipeline. Following group selection, instructions are decoded, renamed and dispatched into issue buffers.

The issue buffers are a pooled resource, and ready (dependence free) instructions are issued without regard to which thread they are from. Instructions are issued in an approximate oldest-first (FCFS) order. There are eight functional units, so up to eight instructions can issue at time, depending on functional unit availability.

© 2007 by J. E. Smith

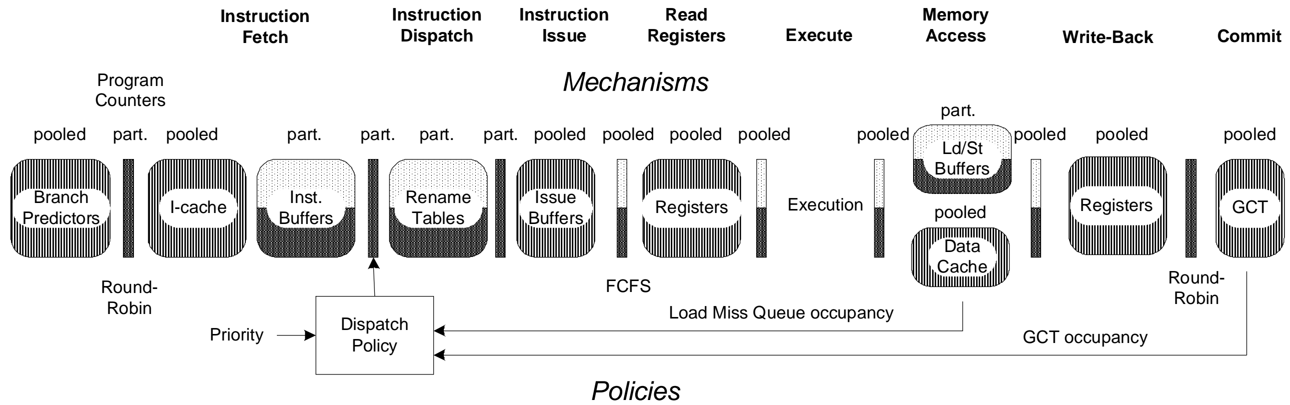


Figure 33. Resource management diagram for IBM Power5.

Issued instructions read operands from the renamed physical registers. The physical registers are pooled, so there is a thread bit appended to the physical register address (two threads require only a single bit thread identifier). The load and store queues are partitioned; there is a significant amount of address checking for memory hazards (as described in Section 3.2.4). This checking is simplified by partitioning the addresses that must be cross-checked.

As shown in Figure 33, the selection policy at the dispatch stage is used as the primary point for resource management. The selection policy is based on priorities, but it also attempts to provide a certain amount of fairness. Each thread has eight software-settable priority levels. All software levels (including user level software) have some control over priorities; however, some of the priority levels can only be set by privileged (system) instructions. Level 0 is assigned when a thread is not running, and levels 1 through 7 apply to running threads. If both threads are in level 1 (the lowest running priority), then the processor throttles back both threads to save power. Figure 34 illustrates relative performance between the two threads for a range of priority levels. Except for at the endpoints of the graph, the x axis indicates the difference in the thread 0 and thread 1 priorities.

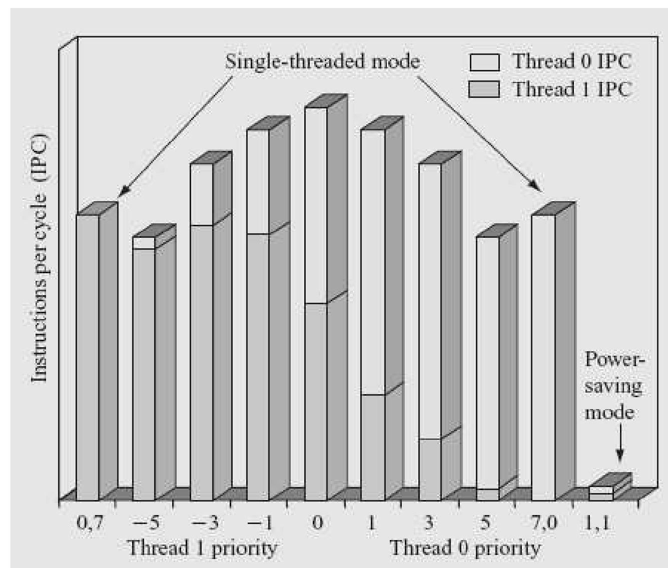


Figure 34. Thread performance for different setting of thread priorities in the IBM Power5.

Software might choose imbalanced priorities in the following situations: 1) a thread may give itself low priority when it is in a spin-loop waiting for a synchronization lock to become available; this might be

© 2007 by J. E. Smith

done by application software, 2) a thread is in an idle loop; this could be done by the OS, 3) a thread must satisfy real-time requirements; in this case, a real-time foreground task may be given priority over a non real-time background thread.

Fairness is provided via what the IBM designers refer to as *dynamic resource-balancing*, which is very close to the philosophy used in the ICOUNT policy described above in Section 3.4.5. There are mechanisms to monitor thread occupancy of the group completion table (GCT) and load miss queue (LMQ). The number of GCT entries for a thread indicates how many outstanding instructions it has, and the number of LMQ entries indicates the number of outstanding data cache misses. For both of these counts, each thread is assigned a threshold. If the threshold is exceeded, then hardware throttles it back.

If a thread exceeds its GCT occupancy threshold, then hardware temporarily reduces its priority until the congestion is cleared. If the thread exceeds it LMQ threshold, then all dispatching from that thread is temporarily inhibited. A larger number of LMQ entries probably means that a high number of issue buffer entries are being tied up by instructions dependent (directly or indirectly) on the load instructions that have missed in the cache.

3.6 References

1. J. W. Rymarczyk, "Coding Guidelines for Pipelined Processors", *Proc. First Int. Symp. on Arch. Support for Programming Languages and Operating Systems ASPLOS-I*, March 1982, pp. 12-19.
2. J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors", *Proceedings of the IEEE*, Dec. 1995, pp. 1609 – 1624.
3. B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System", SPIE Real Time Signal Processing IV, 1981, pp. 241-248.
4. S. Eggers, et al., "Simultaneous Multithreading: A Platform for Next-generation Processors," *IEEE Micro*, September/October 1997.
5. J. Emer, "EV8: The Post-Ultimate Alpha", keynote talk, PACT 2001.
6. D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, Feb. 2002.
7. J. M. Borkenhagen et al., "A Multithreaded PowerPC Processor for Commercial Servers," IBM Journal of Research and Development, 2000.
8. R. Preston et al., Design of an 8-Wide Superscalar RISC Microprocessor with Simultaneous Multithreading, 2002 IEEE Int. Solid State Circuits Conference, paper 20.1.
9. J. D. Davis, et al., Maximizing CMT Throughput with Mediocre Cores, In Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Sep. 2005, pages 51-62.
10. R. Gabor, S. Weiss, A. Mendelson, "Fairness and Throughput in Switch on Event Multithreading", Micro 2006, pp. 149-160, Dec. 2006.
11. L. Hammond, B. A. Nayfeh, K. Olukotun, "A Single Chip Multiprocessor", *IEEE Computer*, pp. 79-85, Sept. 1997.

12. P. Kongetira, et al., Niagara: A 32-way Multithreaded SPARC Processor, IEEE Micro, March/April 2005, pages 21-29.
13. L. Barroso et al. Piranha: A Scalable Architecture Based on Single Chip Multiprocessing, Proc. 27th International Symposium on Computer Architecture, June 2000.
14. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 24th International Symposium on Computer Architecture*, May 1996, pp. 191-202.
15. B. Sinharoy et al., "Power5 System Microarchitecture," *IBM Journal of Research and Development*, July 2005, pp. 505-521.
16. F. J. Cazorla, et al., "Dynamically Controlled Resource Allocation in SMT Processors," Micro 2004, pp. 171-182.
17. R. Kumar, N. Jouppi, D. M. Tullsen, "Conjoined-Core Chip Multiprocessing," Micro 2004, pp. 195-206.
18. E. Tune, et al., "Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy," Micro 2004, pp. 183- 194.
19. H. McGhan, "Niagara 2 Opens the Floodgates," Microprocessor Report, Nov. 2006.
20. S. Eyerman, et al. "A Mechanistic Model for Superscalar Processors," 2007.
21. S. E. Raasch and S. K. Reinhardt, "The Impact of Resource Partitioning on SMT Processors," PACT 2003, pp. 15-25.
22. F. J. Cazorla, et al., "Predictable Performance in SMT Processors: Synergy Between the OS and SMTs," IEEE Trans. Comp, July 2006, pp. 785 - 799.
23. F. J. Cazorla, et al., "QoS for High-Performance SMT Processors in Embedded Systems," IEEE Micro, July-Aug. 2004, pp. 24-31.
24. F. J. Cazorla, et al., "Dcache Warn: an I-fetch Policy to Increase SMT Efficiency," 18th Int. Symp. Parallel and Distributed Processing Symposium, April 2004, pp. 74-83
25. F. J. Cazorla, et al., "Implicit vs. Explicit Resource Allocation in SMT Processors," 2004 Euromicro Symp. on Diginal System Design, Sept. 2004, pp. 44-51.
26. D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading," 2001 Int. Symp. Microarchitecture, Dec. 2001, pp. 318-327.
27. K. Luo, J. Gummaraju, M. Franklin, "Balancing thoughput and fairness in SMT processors," 2001 ISPASS, Nov. 2001, pp. 164-171.
28. P. M. W. Knijnenburg et al., "Branch classification to control instruction fetch in simultaneous multithreaded architectures," 2002 Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, Jan 2002, pp. 67-76
29. R. Jain, C. J. Hughes, S. V. Adve, "Soft real-time scheduling on simultaneous multithreaded processors," 23rd IEEE Real-Time Systems Sympoisum, Dec. 2002,pp. 134-145.
30. A. Snavely and D. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," 9th Int. Symp. on Arch. Support for Programming Languages and Operating Systems, Oct. 2000, pp. 234-244.

31. A. Kumar, "The HP PA-8000 RISC CPU," *IEEE Micro*, March-April 1997, pp. 27-32.

32. G. Chrysos, J. Emer, "Memory Dependence Prediction using Store Sets," 25th Annual International Symposium on Computer Architecture, 1998, pp. 142-153.

33. J. Doweck, "Inside Intel Core Microarchitecture and Smart Memory Access," Intel Corporation Whitepaper, 2006.

34. K. Olukotun, et al., "The Case for a Single-Chip Multiprocessor," ASPLOS-7, October 1996.

35. J. E. Thornton, "Parallel Operation in the CDC 6600," AFIPS Proc. FJCC, pt. 2 vol. 26, 1964, pp. 33-40.

36. E. Rotenberg, et al., "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29), 1996, pp. 24-34.

37. W. A. Wulf, "Compilers and Computer Architecture," *IEEE Computer*, July 1981.

38. C. McNairy and R. Bhatia, "Montecito: A Dual-Core, Dual-Thread Itanium Processor" IEEE Micro. March/April 2005.

39. T. Conte et al., "Optimization of Instruction Fetch Mechanisms for High Issue Rates," 22nd Int. Symp. Computer Architecture, June 1995, pp. 333-344.